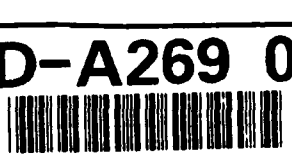
AD-A269 016



TION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	August 1993	Special Technical

4. TITLE AND SUBTITLE

Unreliable Failure Detectors for Reliable Distributed Systems*

5. FUNDING NUMBERS

NAG 2-593

6. AUTHOR(S)

Tushar Deepak Chandra, Sam Toueg

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Sam Toueg, Associate Professor
Department of Computer Science
Cornell University

8. PERFORMING ORGANIZATION REPORT NUMBER

93-1374

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

NASA/ONR

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE
DISTRIBUTION UNLIMITED

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Please see page 1.

93-20467

14. SUBJECT TERMS	15. NUMBER OF PAGES 49
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UNLIMITED

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102

Unreliable Failure Detectors for Reliable Distributed Systems*

Tushar Deepak Chandra**
Sam Toueg

TR 93-1374
(replaces 91-1225)
August 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* Research supported by NSF grants CCr-8901780 and CCR-9102231, DARPA/NASA Ames Grant NAG 2-593, and in part by Grants from IBM and Siemens Corp. A preliminary version of this paper appeared in *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325-340. ACM press, August 1991.
** Also supported by an IBM graduate fellowship.

Unreliable Failure Detectors for Reliable Distributed Systems*

Tushar Deepak Chandra† Sam Toueg

Department of Computer Science
Cornell University
Ithaca, New York 14853
chandra@cs.cornell.edu sam@cs.cornell.edu

Abstract

It is well-known that *Consensus*, a fundamental problem of fault-tolerant distributed computing, cannot be solved in asynchronous systems with crash failures. This impossibility result stems from the lack of reliable failure detection in such systems. To circumvent such impossibility results, we introduce the concept of *unreliable failure detectors* that can make mistakes, and study the problem of using them to solve Consensus.

We characterize unreliable failure detectors by two types of properties: *completeness* and *accuracy*. Informally, *completeness* requires that the failure detector eventually suspects every process that actually crashes, while *accuracy* restricts the mistakes that it can make. We define a hierarchy of failure detectors based on the strength of their accuracy. We determine which failure detectors in this hierarchy can be used to solve Consensus despite any number of crashes, and which ones require a majority of correct processes.

We show that Consensus can be solved with "weak" failure detectors, i.e., failure detectors that make an infinite number of mistakes. This leads to the following question: What is the "weakest" failure detector for solving Consensus? In a companion paper, we show that $\diamond\mathcal{W}$, one of the failure detector that we consider here, is the weakest failure detector for solving Consensus in asynchronous systems.

In this paper, we show that Consensus and Atomic Broadcast are reducible to each other in asynchronous systems. Thus, all our results apply to Atomic Broadcast as well.

*Research supported by NSF grants CCR-8901780 and CCR-9102231, DARPA/NASA Ames Grant NAG-2-593, and in part by Grants from IBM and Siemens Corp. A preliminary version of this paper appeared in *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM press, August 1991.

†Also supported by an IBM graduate fellowship.

1 Introduction

The design and verification of fault-tolerant distributed applications is widely viewed as a complex endeavour. In recent years, several paradigms have been identified which simplify this task. Key among these are *Consensus* and *Atomic Broadcast*. Roughly speaking, Consensus allows processes to reach a common decision, which depends on their initial inputs, despite failures. Consensus algorithms can be used to solve many problems that arise in practice, such as electing a leader or agreeing on the value of a replicated sensor. Atomic Broadcast allows processes to reliably broadcast messages, so that they agree on the set of messages they deliver and the order of message deliveries. Applications based on these paradigms include SIFT [WLG+78], State Machines [Lam78, Sch90], Isis [BJ87, BCJ+90], Psync [PBS89], Amoeba [Mul87], Delta-4 [Pow91], Transis [ADKM91], HAS [Cri87], FAA [CDD90], and Atomic Commitment.

Given their wide applicability, Consensus and Atomic Broadcast have been extensively studied by both theoretical and experimental researchers for over a decade. In this paper, we focus on solutions to Consensus and Atomic Broadcast in the asynchronous model of distributed computing. Informally, a distributed system is *asynchronous* if there is no bound on message delay, clock drift, or the time necessary to execute a step. Thus, to say that a system is asynchronous is to make *no* timing assumptions whatsoever. This model is attractive and has recently gained much currency for several reasons: It has simple semantics; applications programmed on the basis of this model are easier to port than those incorporating specific timing assumptions; and in practice, variable or unexpected workloads are sources of asynchrony—thus synchrony assumptions are at best probabilistic.

Although the asynchronous model of computation is attractive for the reasons outlined above, it is well known that Consensus and Atomic Broadcast cannot be solved deterministically in an asynchronous system that is subject to even a single *crash* failure [FLP85, DDS87].[1] Essentially, the impossibility results for Consensus and Atomic Broadcast stem from the inherent difficulty of determining whether a process has actually crashed or is only "very slow".

To circumvent these impossibility results, previous research focused on the use of randomization techniques [CD89], the definition of some weaker problems and their solutions [DLP+86, ABD+87, BW87, BMZ88], or the study of several models of *partial synchrony* [DDS87, DLS88]. Nevertheless, the impossibility of deterministic solutions to many agreement problems (such as Consensus and Atomic Broadcast) remains a major obstacle to the use of the asynchronous model of computation for fault-tolerant distributed computing.

In this paper, we propose an alternative approach to circumvent such impossibility results, and to broaden the applicability of the asynchronous model of computation. Since impossibility results for asynchronous systems stem from the inherent difficulty of

[1] Roughly speaking, a crash failure occurs when a process that has been executing correctly, stops prematurely. Once a process crashes, it does not recover.

determining whether a process has actually crashed or is only "very slow", we propose to augment the asynchronous model of computation with a model of an external failure detection mechanism that can make mistakes. In particular, we model the concept of *unreliable failure detectors* for systems with *crash* failures.

We consider *distributed* failure detectors: each process has access to a local *failure detector module*. Each local module monitors a subset of the processes in the system, and maintains a list of those that it currently suspects to have crashed. We assume that each failure detector module can make mistakes by erroneously adding processes to its list of suspects: i.e, it can suspect that a process p has crashed even though p is still running. If this module later believes that suspecting p was a mistake, it can remove p from its list. Thus, each module may repeatedly add and remove processes from its list of suspects. Furthermore, at any given time the failure detector modules at two different processes may have different lists of suspects.

It is important to note that the mistakes made by an unreliable failure detector should not prevent any correct process from behaving according to specification even if that process is (erroneously) suspected to have crashed by all the other processes. For example, consider an algorithm that uses a failure detector to solve Atomic Broadcast in an asynchronous system. Suppose all the failure detector modules wrongly (and permanently) suspect that correct process p has crashed. The Atomic Broadcast algorithm must still ensure that p delivers the same set of messages, in the same order, as all the other correct processes. Furthermore, if p broadcasts a message m, all correct processes must deliver m.[2]

We define failure detectors in terms of *abstract* properties as opposed to giving specific *implementations*; the hardware or software implementation of failure detectors is not the concern of this paper. This approach allows us to design applications and prove their correctness relying solely on these properties, without referring to low-level network parameters (such as the exact duration of time-outs that are used to implement failure detectors). This makes the presentation of applications and their proof of correctness more modular. Our approach is well-suited to model many existing systems that decouple the design of fault-tolerant applications from the underlying failure detection mechanisms, such as the *Isis Toolkit* [BCJ+90] for asynchronous fault-tolerant distributed computing.

We characterize a failure detector by specifying the *completeness property* and *accuracy property* that it must satisfy. Informally, *completeness* requires that the failure detector eventually suspects every process that actually crashes,[3] while *accuracy* restricts the mistakes that a failure detector can make. We define two completeness and four accuracy properties, which gives rise to eight failure detectors, and consider the problem

[2] A different approach was taken by the Isis system [RB91]: a correct process that is wrongly suspected to have crashed, is forced to leave the system. In other words, the Isis failure detector forces the system to conform to its view. To applications such a failure detector makes no mistakes. For a more detailed discussion on this, see Section 8.3.

[3] In this introduction, we say that the failure detector suspects that a process p has crashed if *any* local failure detector module suspects that p has crashed.

of solving Consensus using each one of them.[4]

To do so, we first introduce the concept of "reducibility" among failure detectors. Informally, a failure detector $\mathcal{D}'$ *is reducible to failure detector* $\mathcal{D}$ if there is a distributed algorithm that can use $\mathcal{D}$ to emulate $\mathcal{D}'$. Given this reduction algorithm, anything that can be done using failure detector $\mathcal{D}'$, can be done using $\mathcal{D}$ instead. Two failure detectors are *equivalent* if they are reducible to each other. Using this concept, we partition our eight failure detectors into four equivalence classes, and consider how to solve Consensus for each class.

We show that only four of our eight failure detectors can be used to solve Consensus in systems in which any number of processes may crash. However, if we assume that a majority of the processes do not crash, then any of our eight failure detectors can be used to solve Consensus. In order to better understand where the majority requirement becomes necessary, we study an infinite hierarchy of failure detectors that contains the eight failure detectors mentioned above, and show exactly where in this hierarchy the majority requirement becomes necessary.

Of special interest is $\diamond\mathcal{W}$, the weakest failure detector considered in this paper. Informally, $\diamond\mathcal{W}$ satisfies the following two properties:

- *Completeness*: There is a time after which every process that crashes is always suspected by some correct process.
- *Accuracy*: There is a time after which some correct process is never suspected by any correct process.

The failure detector $\diamond\mathcal{W}$ can make an infinite number of mistakes: Each local failure detector module of $\diamond\mathcal{W}$ can repeatedly add and then remove *correct* processes from its list of suspects (this reflects the inherent difficulty of determining whether a process or a link is just slow or whether it has crashed). Moreover, some correct processes may be erroneously suspected to have crashed by all the other processes throughout the entire execution.

The two properties of $\diamond\mathcal{W}$ state that eventually something must hold forever; this may appear too strong a requirement to implement in practice. However, when solving a problem that "terminates", such as Consensus, it is not really required that the properties hold *forever*, but merely that they hold for a *sufficiently long time*, i.e., long enough for the algorithm that uses the failure detector to achieve its goal. For instance, in practice our algorithm that solves Consensus using $\diamond\mathcal{W}$ only needs the two properties of $\diamond\mathcal{W}$ to hold for a relatively short period of time. However, in an asynchronous system it is not possible to quantify "sufficiently long", since even a single process step or a single message transmission is allowed to take an arbitrarily long amount of time. Thus, it is convenient to state the properties of $\diamond\mathcal{W}$ in the stronger form given above.

[4] We later show that Consensus and Atomic Broadcast are *equivalent* in asynchronous systems: any Consensus algorithm can be transformed into an Atomic Broadcast algorithm and vice versa. Thus, we can focus on Consensus since all our results will automatically apply to Atomic Broadcast as well.

Another advantage of using $\diamond\mathcal{W}$ (as opposed to stronger failure detectors) is the following. Consider an application that relies on $\diamond\mathcal{W}$ for its correctness. If this application is run in a system in which the failure detector "malfunctions" and fails to meet the specification of $\diamond\mathcal{W}$, then we may lose the *liveness* properties of the application, but its *safety* properties will never be violated.

The failure detector abstraction is a clean extension to the asynchronous model of computation that allows us to solve many problems that are otherwise unsolvable. Naturally, the question arises of how to support such an abstraction in an actual system. Since we specify failure detectors in terms of abstract properties, we are not committed to a particular implementation. For instance, one could envision specialised hardware to support this abstraction. However, most implementations of failure detectors are based on time-out mechanisms. For the purpose of illustration, we now outline one such implementation of $\diamond\mathcal{W}$.

Informally, if a process times-out on some process q, it adds q to its list of suspects, and it broadcasts a message to all processes (including q) with this information. Any process that receives this broadcast adds q to its list of suspects. If q has not crashed, it broadcasts a refutation. If a process receives q's refutation, it removes q from its list of suspects.

In the *purely* asynchronous system, this scheme does not implement $\diamond\mathcal{W}$:[5] an unbounded sequence of premature time-outs (with corresponding refutations) may cause every correct process to be repeatedly added and then removed from every correct process' list of suspects, thereby violating the accuracy property of $\diamond\mathcal{W}$. Nevertheless, in many practical systems, one can choose the time-out periods so that eventually there are no premature time-outs on at least one correct process p. This gives the accuracy property of $\diamond\mathcal{W}$: there is a time after which p is permanently removed from all the lists of suspects. Recall that, in practice, it is not necessary for this to hold permanently; it is sufficient that it holds "long enough" for the application using the failure detector to complete its task. Accordingly, it is not necessary for the premature time-outs on p to cease permanently: it is sufficient that they cease for "long enough".

Having made the point that $\diamond\mathcal{W}$ can be implemented in practical systems using time-outs, we reiterate that all reasoning about failure detectors (and algorithms that use them) should be done in terms of their abstract properties and not in terms of any particular implementation. This is an important feature of this approach, and the reader should refrain from thinking of failure detectors in terms of specific time-out mechanisms.

The failure detection information provided by $\diamond\mathcal{W}$, the weakest failure detector considered in this paper, is *sufficient* to solve Consensus. But is it *necessary*? In other words, is it possible to solve Consensus with a failure detector that provides less information about failures than $\diamond\mathcal{W}$? Indeed, what it is the "weakest" failure detector for solving Consensus? In [CHT92], we show that $\diamond\mathcal{W}$ *is* the weakest failure detector that can be

[5] Indeed, no scheme could implement $\diamond\mathcal{W}$ in the purely asynchronous system: as we show in Section 6.2, such an implementation could be used to solve Consensus in such a system, contradicting the impossibility result of [FLP85].

used to solve Consensus in asynchronous systems (with a majority of correct processes). More precisely, we show how to emulate $\diamond\mathcal{W}$ using *any* failure detector $\mathcal{D}$ that can be used to solve Consensus. Thus, in a precise sense, $\diamond\mathcal{W}$ is necessary and sufficient for solving Consensus in asynchronous systems (with a majority of correct processes). This result is further evidence to the importance of $\diamond\mathcal{W}$ for fault-tolerance in asynchronous distributed computing.

In our discussion so far, we focused on the Consensus problem. In Section 7, we show that Consensus is equivalent to Atomic Broadcast in asynchronous systems with crash failures. This is shown by reducing each problem to the other.[6] In other words, a solution for one automatically yields a solution for the other. Both reductions apply to any asynchronous system (in particular, they do *not* require the assumption of a failure detector). Thus, Atomic Broadcast can be solved using the unreliable failure detectors described in this paper. Furthermore, $\diamond\mathcal{W}$ is the weakest failure detector that can be used to solve Atomic Broadcast.

A different tack on circumventing the unsolvability of Consensus is pursued in [DDS87] and [DLS88]. The approach of those papers is based on the observation that between the completely synchronous and completely asynchronous models of distributed systems there lie a variety of intermediate *partially synchronous* models. In particular, those two papers consider 34 different models of partial synchrony and for each model determine whether or not Consensus can be solved. In this paper, we argue that partial synchrony assumptions can be encapsulated in the unreliability of the failure detector. For example, we show how to implement one of our failure detectors (which is stronger than $\diamond\mathcal{W}$), in the models of partial synchrony considered in [DLS88]. This immediately implies that Consensus and Atomic Broadcast can be solved in these models. Thus, our approach can be used to unify several seemingly unrelated models of partial synchrony.[7]

As we argued earlier, using the asynchronous model of computation is highly desirable in many applications: it results in code that is simple, portable and robust. However, the fact that fundamental problems such as Consensus and Atomic Broadcast have no (deterministic) solutions in this model is a major obstacle to its use in fault-tolerant distributed computing. Our model of unreliable failure detectors provides a natural and simple extension of the asynchronous model of computation, in which Consensus and Atomic Broadcast can be solved deterministically. Thus, this extended model retains the advantages of asynchrony without inheriting its disadvantages. We believe that this approach is an important contribution towards bridging the gap between known theoretical impossibility results and the need for fault-tolerant solutions in real systems.

The remainder of this paper is organised as follows. In Section 2, we describe our model and introduce eight failure detectors in terms of their abstract properties. In Section 3, we show that these eight failure detectors fall into four equivalence classes—this allows us to focus on four failure detectors rather than eight. In Section 4, we

[6] They are actually equivalent even in asynchronous systems with *arbitrary*, i.e., "Byzantine", failures. However, that reduction is more complex and is omitted from this paper.

[7] For a more detailed discussion on this, see Section 8.

present *Reliable Broadcast*, a communication primitive that several of our algorithms use. In Section 5, we define the Consensus problem. In Section 6, we show how to solve Consensus for each one of the four equivalence classes of failure detectors. In Section 7, we show that Consensus and Atomic Broadcast are equivalent to each other in asynchronous systems. In Section 8, we discuss related work, and in particular, we describe an implementation of an unreliable failure detector that is more powerful than $\diamond\mathcal{W}$, in several models of partial synchrony. In the Appendix we define an infinite hierarchy of failure detectors, and determine exactly where in this hierarchy a majority of correct processes is required to solve Consensus.

2 The model

We consider *asynchronous* distributed systems in which there is no bound on message delay, clock drift, or the time necessary to execute a step. Our model of asynchronous computation with failure detection is patterned after the one in [FLP85]. The system consists of a set of n *processes*, $\Pi = \{p_1, p_2, \ldots, p_n\}$. Every pair of processes is connected by a reliable communication channel.

To simplify the presentation of our model, we assume the existence of a discrete global clock. This is merely a fictional device: the processes do not have access to it. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers.

2.1 Failures and failure patterns

Processes can fail by *crashing*, i.e., by prematurely halting. A *failure pattern* F is a function from $\mathcal{T}$ to 2^{Π}, where $F(t)$ denotes the set of processes that have crashed through time t. Once a process crashes, it does not "recover", i.e., $\forall t : F(t) \subseteq F(t+1)$. We define $\mathit{crashed}(F) = \bigcup_{t \in \mathcal{T}} F(t)$ and $\mathit{correct}(F) = \Pi - \mathit{crashed}(F)$. If $p \in \mathit{crashed}(F)$ we say p *crashes in* F and if $p \in \mathit{correct}(F)$ we say p *is correct in* F. We consider only failure patterns F such that at least one process is correct, i.e., $\mathit{correct}(F) \neq \emptyset$.

2.2 Failure detectors

Each failure detector module outputs the set of processes that it currently suspects to have crashed.[8] A *failure detector history* H is a function from $\Pi \times \mathcal{T}$ to 2^{Π}. $H(p,t)$ is the value of the failure detector module of process p at time t. If $q \in H(p,t)$, we say that p *suspects* q *at time* t *in* H. We omit references to H when it is obvious from the context. Note that the failure detector modules of two different processes need not agree on the list of processes that are suspected to have crashed, i.e., if $p \neq q$ then $H(p,t) \neq H(q,t)$ is possible.

[8] In [CHT92] we study a more general class of failure detectors: their modules can output values from an arbitrary range.

Informally, a failure detector $\mathcal{D}$ provides (possibly incorrect) information about the failure pattern F that occurs in an execution. Formally, *failure detector* $\mathcal{D}$ is a function that maps each failure pattern F to a set of failure detector histories $\mathcal{D}(F)$. This is the set of all failure detector histories that could occur in executions with failure pattern F and failure detector $\mathcal{D}$.[9]

In this paper, we do not define failure detectors in terms of specific *implementations*. Such implementations would have to refer to low-level network parameters, such as the network topology, the message delays, and the accuracy of the local clocks. To avoid this problem, we specify a failure detector in terms of two *abstract properties* that it must satisfy: *completeness* and *accuracy*. This allows us to design applications and prove their correctness relying solely on these properties.

2.3 Failure detector properties

2.3.1 Completeness

We consider two completeness properties:

- *Strong completeness*: Eventually every process that crashes is permanently suspected by *every* correct process. Formally, $\mathcal{D}$ satisfies strong completeness if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \forall q \in \mathit{correct}(F), \forall t' \geq t : p \in H(q, t')$$

- *Weak completeness*: Eventually every process that crashes is permanently suspected by *some* correct process. Formally, $\mathcal{D}$ satisfies weak completeness if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall p \in \mathit{crashed}(F), \exists q \in \mathit{correct}(F), \forall t' \geq t : p \in H(q, t')$$

However, completeness by itself is not a useful property. To see this, consider a failure detector which causes every process to permanently suspect every other process in the system. Such a failure detector trivially satisfies strong completeness but is clearly useless since it provides no information about failures. To be useful, a failure detector must also satisfy some accuracy property that restricts the *mistakes* that it can make. We now consider such properties.

2.3.2 Accuracy

Consider the following two accuracy properties:

[9] In general, there are many executions with the same failure pattern F (e.g, these executions may differ by the pattern of their message exchange). For each such execution, $\mathcal{D}$ may give a different failure detector history.

- *Strong accuracy.* No process is suspected before it crashes. Formally, $\mathcal{D}$ satisfies strong accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \mathcal{T}, \forall p, q \in \Pi - F(t) : p \notin H(q,t)$$

Since it is difficult (if not impossible) to achieve strong accuracy in many practical systems, we also define:

- *Weak accuracy.* Some correct process is never suspected. Formally, $\mathcal{D}$ satisfies weak accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists p \in \mathit{correct}(F), \forall t \in \mathcal{T}, \forall q \in \Pi - F(t) : p \notin H(q,t)$$

Even weak accuracy guarantees that at least one correct process is *never* suspected. Since this type of accuracy may be difficult to achieve, we consider failure detectors that may suspect *every* process at one time or another. Informally, we only require that strong accuracy or weak accuracy are *eventually* satisfied. The resulting properties are called *eventual strong accuracy* and *eventual weak accuracy*, respectively.

For example, eventual strong accuracy requires that there is a time after which strong accuracy holds. Formally, $\mathcal{D}$ satisfies eventual strong accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall t' \geq t, \forall p, q \in \Pi - F(t') : p \notin H(q,t')$$

An observation is now in order. Since all faulty processes will crash after some finite time, we have:

$$\forall F, \exists t \in \mathcal{T}, \forall t' \geq t : \Pi - F(t') = \mathit{correct}(F)$$

Thus, an equivalent and simpler formulation of eventual strong accuracy is:

- *Eventual strong accuracy.* There is a time after which correct processes are not suspected by any correct process. Formally, $\mathcal{D}$ satisfies eventual strong accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \forall t' \geq t, \forall p, q \in \mathit{correct}(F) : p \notin H(q,t')$$

Similarly, we specify eventual weak accuracy as follows:

- *Eventual weak accuracy.* There is a time after which some correct process is never suspected by any correct process. Formally, $\mathcal{D}$ satisfies eventual weak accuracy if:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \mathcal{T}, \exists p \in \mathit{correct}(F), \forall t' \geq t, \forall q \in \mathit{correct}(F) : p \notin H(q,t')$$

We will refer to eventual strong accuracy and eventual weak accuracy as *eventual accuracy* properties, and strong accuracy and weak accuracy as *perpetual accuracy* properties.

Completeness	Accuracy			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect FD* $\mathcal{P}$	*Strong FD* $\mathcal{S}$	*Eventually Perfect FD* $\diamond\mathcal{P}$	*Eventually Strong FD* $\diamond\mathcal{S}$
Weak	$\mathcal{Q}$	*Weak FD* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak FD* $\diamond\mathcal{W}$

Figure 1: Some failure detector specifications based on accuracy and completeness.

2.4 Some failure detector definitions

A failure detector can be specified by stating the completeness property and the accuracy property that it must satisfy. Combining the two completeness properties with the four accuracy properties that we defined in the previous section gives rise to the eight different failure detectors defined in Figure 1. For example, we say that a failure detector is *Eventually Strong* if it satisfies strong completeness and eventual weak accuracy. We denote such a failure detector by $\diamond\mathcal{S}$.

2.5 Algorithms and runs

In this paper, we focus on algorithms that use unreliable failure detectors. To describe such algorithms, we only need informal definitions of algorithms and runs, based on the formal definitions given in [CHT92].[10]

An algorithm A is a collection of n deterministic automata, one for each process in the system. Computation proceeds in *steps* of A. In each step, a process (1) may receive a message that was sent to it, (2) queries its failure detector module, (3) undergoes a state transition, and (4) may send a message. Since we model asynchronous systems, messages may experience arbitrary (but finite) delays. Furthermore, there is no bound on relative process speeds.

Informally, a *run of algorithm A using a failure detector $\mathcal{D}$* is a tuple $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ where F is a failure pattern, $H_{\mathcal{D}} \in \mathcal{D}(F)$ is a history of failure detector $\mathcal{D}$ for failure pattern F, I is an initial configuration of A, S is an infinite sequence of steps of A, and T is a list of increasing time values indicating when each step in S occurred. A run must satisfy certain well-formedness and fairness properties. In particular, (1) a process cannot take a step after it crashes, (2) when a process takes a step and queries its failure detector module, it gets the current value output by its local failure detector module, and (3) every process that is correct in F takes an infinite number of steps in S and

[10] Formal definitions are necessary in [CHT92] to prove a subtle lower bound.

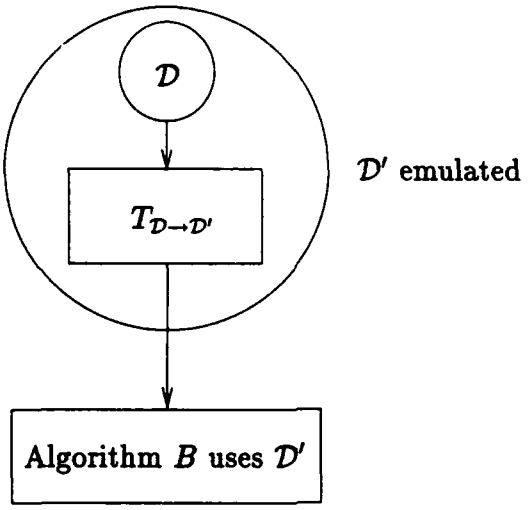


Figure 2: Transforming $\mathcal{D}$ into $\mathcal{D}'$

eventually receives every message sent to it.

We use the following notation. Let v be a variable in algorithm A. We denote by v_p process p's copy of v. The history of v in run R is denoted by v^R, i.e., $v^R(p,t)$ is the value of v_p at time t in run R. We denote by $\mathcal{D}_p$ process p's local failure detector module. Thus, the value of $\mathcal{D}_p$ at time t in run $R = \langle F, H_\mathcal{D}, I, S, T\rangle$ is $H_\mathcal{D}(p,t)$.

2.6 Reducibility

We now define what it means for an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ to transform a failure detector $\mathcal{D}$ into another failure detector $\mathcal{D}'$. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ to maintain a variable $output_p$ at every process p. This variable, reflected in the local state of p, emulates the output of $\mathcal{D}'$ at p. Algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ *transforms* $\mathcal{D}$ *into* $\mathcal{D}'$ if and only if for every run $R = \langle F, H_\mathcal{D}, I, S, T\rangle$ of $T_{\mathcal{D}\to\mathcal{D}'}$ using $\mathcal{D}$, $output^R \in \mathcal{D}'(F)$.

Given the reduction algorithm $T_{\mathcal{D}\to\mathcal{D}'}$, anything that can be done using failure detector $\mathcal{D}'$, can be done using $\mathcal{D}$ instead. To see this, suppose a given algorithm B requires failure detector $\mathcal{D}'$, but only $\mathcal{D}$ is available. We can still execute B as follows. Concurrently with B, processes run $T_{\mathcal{D}\to\mathcal{D}'}$ to transform $\mathcal{D}$ into $\mathcal{D}'$. We modify Algorithm B at process p as follows: whenever p is required to query its failure detector module, p reads the current value of $output_p$ (which is concurrently maintained by $T_{\mathcal{D}\to\mathcal{D}'}$) instead. This is illustrated in Figure 2.

Intuitively, since $T_{\mathcal{D}\to\mathcal{D}'}$ is able to use $\mathcal{D}$ to emulate $\mathcal{D}'$, $\mathcal{D}$ provides at least as much information about process failures as $\mathcal{D}'$ does. Thus, if there is an algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ that transforms $\mathcal{D}$ into $\mathcal{D}'$, we write $\mathcal{D} \succeq \mathcal{D}'$ and say that $\mathcal{D}'$ *is reducible to* $\mathcal{D}$; we also say

that $\mathcal{D}'$ *is weaker than* $\mathcal{D}$. If $\mathcal{D} \succeq \mathcal{D}'$ and $\mathcal{D}' \succeq \mathcal{D}$, we write $\mathcal{D} \cong \mathcal{D}'$ and say that $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent.*

Note that, in general, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ need not emulate *all* the failure detector histories of $\mathcal{D}'$; what we do require is that all the failure detector histories it emulates be histories of $\mathcal{D}'$.

Consider the "identity" transformation $T_{\mathcal{D}\rightarrow\mathcal{D}}$ in which each process p periodically writes the current value output by its local failure detector module into $output_p$. The following is immediate from $T_{\mathcal{D}\rightarrow\mathcal{D}}$ and the definition of reducibility.

Observation 1: $\mathcal{P} \succeq \mathcal{Q}$, $\mathcal{S} \succeq \mathcal{W}$, $\Diamond\mathcal{P} \succeq \Diamond\mathcal{Q}$, $\Diamond\mathcal{S} \succeq \Diamond\mathcal{W}$.

3 From weak completeness to strong completeness

In Figure 3, we give a reduction algorithm $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ that transforms any given failure detector $\mathcal{D}$ that satisfies weak completeness, into a failure detector $\mathcal{D}'$ that satisfies strong completeness. Furthermore, for each failure detector $\mathcal{D}$ defined in Figure 1, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ gives a failure detector $\mathcal{D}'$ that has the same accuracy property as $\mathcal{D}$. Roughly speaking, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ strengthens the completeness property while preserving accuracy.

This result allows us to focus on the failure detectors that are defined in the first row of Figure 1, i.e., those with strong completeness. This is because, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ (together with Observation 1) shows that every failure detector in the second row of Figure 1 is actually *equivalent* to the corresponding failure detector above it in that figure.

Informally, $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ works as follows. Every process p periodically sends $(p, suspects_p)$—where $suspects_p$ denotes the set of processes that p suspects according to its local failure detector module—to all the processes. When a process q receives a message of the form $(p, suspects_p)$, it adds $suspects_p$ to $output_q$ and removes p from $output_q$.

Let $R = \langle F, H_{\mathcal{D}}, I, S, T\rangle$ be an arbitrary run of $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ using failure detector $\mathcal{D}$. In the following, the run R and its failure pattern F are fixed. Thus, when we say that a process crashes we mean that it crashes in F. Similarly, when we say that a process is correct, we mean that it is correct in F. We will show that $output^R$ satisfies the following properties:

P1 : *(Transforming weak completeness into strong completeness)* Let p be any process that crashes. If eventually *some* correct process permanently suspects p in $H_{\mathcal{D}}$, then eventually *all* correct processes permanently suspect p in $output^R$. More formally:

$$\begin{aligned} &\forall p \in crashed(F): \\ &\qquad \exists t \in \mathcal{T}, \exists q \in correct(F), \forall t' \geq t : p \in H_{\mathcal{D}}(q, t') \\ \Rightarrow \quad &\qquad \exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \in output^R(q, t') \end{aligned}$$

P2 : *(Preserving perpetual accuracy)* Let p be any process. If no process suspects p in $H_{\mathcal{D}}$ before time t, then no process suspects p in $output^R$ before time t. More formally:

Every process p executes the following:

$output_p \leftarrow \emptyset$

cobegin
|| *Task 1:* **repeat forever**
  $\{p$ *queries its local failure detector module* $\mathcal{D}_p\}$
  $suspects_p \leftarrow \mathcal{D}_p$
  send $(p, suspects_p)$ to all

|| *Task 2:* **when** receive $(q, suspects_q)$ for some q
  $output_p \leftarrow (output_p \cup suspects_q) - \{q\}$
coend

Figure 3: $T_{\mathcal{D}\rightarrow\mathcal{D}'}$: From Weak Completeness to Strong Completeness

$$\forall p \in \Pi, \forall t \in \mathcal{T}:$$
$$\forall t' < t, \forall q \in \Pi - F(t') : p \notin H_{\mathcal{D}}(q, t')$$
$$\Rightarrow \quad \forall t' < t, \forall q \in \Pi - F(t') : p \notin output^R(q, t')$$

P3 : *(Preserving eventual accuracy)* Let p be any correct process. If there is a time after which no correct process suspects p in $H_{\mathcal{D}}$, then there is a time after which no correct process suspects p in $output^R$. More formally:

$$\forall p \in correct(F):$$
$$\exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \notin H_{\mathcal{D}}(q, t')$$
$$\Rightarrow \quad \exists t \in \mathcal{T}, \forall q \in correct(F), \forall t' \geq t : p \notin output^R(q, t')$$

Lemma 2: $T_{\mathcal{D}\rightarrow\mathcal{D}'}$ satisfies P1.

PROOF: Let p be any process that crashes. Suppose that there is a time t after which some correct process q permanently suspects p in $H_{\mathcal{D}}$. We must show that there is a time after which every correct process suspects p in $output^R$.

Since p crashes, there is a time t' after which no process receives a message from p. Consider the execution of Task 1 by process q after time $t_p = \max(t, t')$. Process q sends a message of the type $(q, suspects_q)$ with $p \in suspects_q$ to all processes. Eventually, every correct process receives $(q, suspects_q)$ and adds p to $output$ (see Task 2). Since no correct process receives any messages from p after time t' and $t_p \geq t'$, no correct process

removes p from *output* after time t_p. Thus, there is a time after which every correct process permanently suspects p in $output^R$. □

Lemma 3: $T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P2.

PROOF: Let p be any process. Suppose that there is a time t before which no process suspects p in $H_{\mathcal{D}}$. No process sends a message of the type $(-, suspects)$ with $p \in suspects$ before time t. Thus, no process q adds p to $output_q$ before time t. □

Lemma 4: $T_{\mathcal{D}\to\mathcal{D}'}$ satisfies P3.

PROOF: Let p be any correct process. Suppose that there is a time t after which no correct process suspects p in $H_{\mathcal{D}}$. Thus, all processes that suspect p after time t eventually crash. Thus, there is a time t' after which no correct process receives a message of the type $(-, suspects)$ with $p \in suspects$.

Let q be any correct process. We must show that there is a time after which q does not suspect p in $output^R$.

Consider the execution of Task 1 by process p after time t'. Process p sends a message $m = (p, suspects_p)$ to q. When q receives m, it removes p from $output_q$ (see Task 2). Since q does not receive any messages of the type $(-, suspects)$ with $p \in suspects$ after time t', q does not add p to $output_q$ after time t'. Thus, there is a time after which q does not suspect p in $output^R$. □

Theorem 5: $T_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{Q}$ into $\mathcal{P}$, $\mathcal{W}$ into $\mathcal{S}$, $\diamond\mathcal{Q}$ into $\diamond\mathcal{P}$, and $\diamond\mathcal{W}$ into $\diamond\mathcal{S}$.

PROOF: By Lemma 2, $T_{\mathcal{D}\to\mathcal{D}'}$ transforms $\mathcal{Q}$, $\mathcal{W}$, $\diamond\mathcal{Q}$, and $\diamond\mathcal{W}$, into failure detectors that satisfy strong completeness. By Lemma 3, $T_{\mathcal{D}\to\mathcal{D}'}$ preserves the strong accuracy of $\mathcal{Q}$ and the weak accuracy of $\mathcal{W}$. By Lemma 4, $T_{\mathcal{D}\to\mathcal{D}'}$ preserves the eventual strong accuracy of $\diamond\mathcal{Q}$ and the eventual weak accuracy of $\diamond\mathcal{W}$. The theorem immediately follows. □

By Theorem 5 and Observation 1, we have:

Corollary 6: $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamond\mathcal{P} \cong \diamond\mathcal{Q}$, and $\diamond\mathcal{S} \cong \diamond\mathcal{W}$.

4 Reliable Broadcast

We now define Reliable Broadcast, a communication primitive that we often use in our algorithms. Informally, Reliable Broadcast guarantees that (1) all correct processes deliver the same set of messages, (2) all messages broadcast by correct processes are delivered, and (3) no spurious messages are ever delivered. Formally, Reliable Broadcast is defined in terms of two primitives, *R-broadcast*(m) and *R-deliver*(m) where m is a message drawn from a set of possible messages. When a process executes *R-broadcast*(m), we say that it *R-broadcasts* m, and when a process executes *R-deliver*(m), we say that

Every process p executes the following:

To execute R-broadcast(m):
    send m to all (including p)

R-deliver(m) *occurs as follows:*
    when receive m for the first time
        if *sender*(m) $\neq p$ **then** send m to all
        R-deliver(m)

Figure 4: Reliable Broadcast by message diffusion

it *R-delivers* m. Reliable Broadcast satisfies the following three properties:[11]

Validity: If a correct process R-broadcasts a message m, then all correct processes eventually R-deliver m.

Agreement: If a correct process R-delivers a message m, then all correct processes eventually R-deliver m.

Uniform integrity: For any message m, each process R-delivers m at most once, and only if m was R-broadcast by some process.

In Figure 4, we give a simple Reliable Broadcast algorithm for asynchronous systems. Informally, when a process receives a message for the first time, it relays the message to all processes and then R-delivers it. It is easy to show that this algorithm satisfies validity, agreement and uniform integrity in asynchronous systems with up to $n-1$ crash failures. The proof is obvious and therefore omitted.

5 The Consensus problem

In the Consensus problem, all correct processes propose a value and must reach a unanimous and irrevocable decision on some value that is related to the proposed values [Fis83]. We define the Consensus problem in terms of two primitives, *propose*(v) and *decide*(v), where v is a value drawn from a set of possible proposed values. When a process executes *propose*(v), we say that it *proposes* v; similarly, when a process executes *decide*(v), we say that it *decides* v. The *Consensus* problem is specified as follows:

[11] For simplicity, we assume that each message is unique. In practice, this can be achieved by tagging the identity of the sender and a sequence number on each message.

Termination: Every correct process eventually decides some value.

Uniform validity: If a process decides v, then v was proposed by some process.[12]

Uniform integrity: Every process decides at most once.

Agreement: No two correct processes decide differently.

It has been proved that there is no deterministic algorithm for Consensus in asynchronous systems that are subject to even a single crash failure [FLP85, DDS87]. We now show how to use unreliable failure detectors to solve Consensus in asynchronous systems.

6 Solving Consensus using unreliable failure detectors

We now show how to solve Consensus using each one of the eight failure detectors defined in Figure 1. By Theorem 5, we only need to show how to solve Consensus using the four failure detectors that satisfy strong completeness, namely, $\mathcal{P}$, $\mathcal{S}$, $\diamond\mathcal{P}$, and $\diamond\mathcal{S}$.

Solving Consensus with the Perfect failure detector $\mathcal{P}$ is simple, and is left as an exercise for the reader. In Section 6.1, we give a Consensus algorithm that uses $\mathcal{S}$. In Section 6.2, we give a Consensus algorithm that uses $\diamond\mathcal{S}$. Since $\diamond\mathcal{P} \succeq \diamond\mathcal{S}$, this algorithm also solves Consensus with $\diamond\mathcal{P}$.

The Consensus algorithm that uses $\mathcal{S}$ can tolerate any number of failures. In contrast, the one that uses $\diamond\mathcal{S}$ requires a majority of correct processes. We show that this requirement is necessary even if one uses $\diamond\mathcal{P}$, a failure detector that is stronger than $\diamond\mathcal{S}$. Thus, our algorithm for solving Consensus using $\diamond\mathcal{S}$ (or $\diamond\mathcal{P}$) is optimal with respect to the number of failures that it tolerates.

6.1 Using a Strong failure detector $\mathcal{S}$

Given any Strong failure detector $\mathcal{S}$, the algorithm in Figure 5 solves Consensus in asynchronous systems. This algorithm runs through 3 phases. In Phase 1, processes execute $n-1$ asynchronous rounds (r_p denotes the current round number of process p) during which they broadcast and relay their proposed values. Each process p waits until it receives a round r message from every process that is not in $\mathcal{S}_p$, before proceeding to round $r+1$. Note that it is possible that while p is waiting for a message from q in round r, q is added to $\mathcal{S}_p$. By the above rule, p stops waiting for q's message and proceeds to round $r+1$.

By the end of Phase 2, correct processes agree on a vector based on the proposed values of all processes. The ith element of this vector either contains the proposed value

[12] The validity condition captures the relation between the decision value and the proposed values. Changing this condition results in other types of Consensus [Fis83].

of process p_i or $\perp$. We will show that this vector contains the proposed value of at least one process. In Phase 3, correct processes decide the first non-trivial component of this vector.

Let f denote the maximum number of processes that may crash.[13] Phase 1 of the algorithm consists of $n-1$ rounds, rather than the usual $f+1$ rounds of traditional Consensus algorithms (for synchronous systems). Intuitively, this is because even a *correct* process p may be suspected to have crashed by other processes. In this case, p's messages may be ignored, and p appears to commit "send-omission" failures. Thus, up to $n-1$ processes may appear to commit such failures (rather than f). Note that because $\mathcal{S}$ satisfies weak accuracy (namely, some correct process is never suspected), the maximum number of processes that may fail or appear to fail is $n-1$ rather than n.

$V_p[q]$ denotes p's current estimate of q's proposed value. $\Delta_p[q] = v_q$ at the end of round r if and only if p receives v_q, the value proposed by q, for the first time in round r.

Let $R = \langle F, H_{\mathcal{S}}, I, S, T \rangle$ be any run of the algorithm in Figure 5 using $\mathcal{S}$ in which all correct processes propose a value. We have to show that termination, uniform validity, agreement and uniform integrity hold.

Lemma 8: For all p and q, and in all phases, $V_p[q]$ is either v_q or $\perp$.

PROOF: Obvious from the algorithm in Figure 5. □

Lemma 9: Every correct process eventually reaches Phase 3.

PROOF: [*sketch*] The only way a correct process p can be prevented from reaching Phase 3 is by blocking forever at one of the two **wait** statements (in Phase 1 and 2, respectively). This can happen only if p is waiting forever for a message from a process q and q never joins $\mathcal{S}_p$. There are two cases to consider:

1. q crashes. Since $\mathcal{S}$ satisfies strong completeness, there is a time after which $q \in \mathcal{S}_p$.
2. q does not crash. In this case, we can show (by an easy but tedious induction on the round number) that q eventually sends the message p is waiting for.

In both cases p is not blocked forever and reaches Phase 3. □

Since $\mathcal{S}$ satisfies weak accuracy there is a correct process c that is never suspected by any process, i.e., $\forall t \in \mathcal{T}, \forall p \in \Pi - F(t) : c \notin H_{\mathcal{S}}(p,t)$. Let Π_1 denote the set of processes that complete all $n-1$ rounds of Phase 1, and Π_2 denote the set of processes that complete Phase 2. We say $V_p \leq V_q$ if and only if for all $k \in \Pi$, $V_p[k]$ is either $V_q[k]$ or $\perp$.

Lemma 10: In every round r, $1 \leq r \leq n-1$, all processes $p \in \Pi_1$ receive (r, Δ_c, c) from process c, i.e., (r, Δ_c, c) is in $msgs_p[r]$.

[13] In the literature, t is often used instead of f, the notation adopted here. In this paper, we reserve t to denote real-time.

Every process p executes the following:

procedure *propose*(v_p)
  $V_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$ {*p's estimate of the proposed values*}
  $V_p[p] \leftarrow v_p$
  $\Delta_p \leftarrow V_p$

  Phase 1: {*asynchronous rounds* r_p, $1 \leq r_p \leq n-1$}
    for $r_p \leftarrow 1$ to $n-1$
      send (r_p, Δ_p, p) to all
      wait until $[\forall q:$ received (r_p, Δ_q, q) **or** $q \in \mathcal{S}_p]$
        {*Query the failure detector*}
      $msgs_p[r_p] \leftarrow \{(r_p, \Delta_q, q) \mid$ received $(r_p, \Delta_q, q)\}$
      $\Delta_p \leftarrow \langle \perp, \perp, \ldots, \perp \rangle$
      for $k \leftarrow 1$ to n
        if $V_p[k] = \perp$ **and** $\exists (r_p, \Delta_q, q) \in msgs_p[r_p]$ with $\Delta_q[k] \neq \perp$ **then**
          $V_p[k] \leftarrow \Delta_q[k]$
          $\Delta_p[k] \leftarrow \Delta_q[k]$

  Phase 2: send V_p to all
    wait until $[\forall q:$ received V_q **or** $q \in \mathcal{S}_p]$ {*Query the failure detector*}
    $lastmsgs_p \leftarrow \{V_q \mid$ received $V_q\}$
    for $k \leftarrow 1$ to n
      if $\exists V_q \in lastmsgs_p$ with $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

  Phase 3: *decide*(first non-$\perp$ component of V_p)

Figure 5: Solving Consensus using $\mathcal{S}$.

PROOF: Since $p \in \Pi_1$, p completes all $n-1$ rounds of Phase 1. At each round r, since $c \notin \mathcal{S}_p$, p waits for and receives the message (r, Δ_c, c) from c. □

Lemma 11: For all $p \in \Pi_1$, $V_c \leq V_p$ at the end of Phase 1.

PROOF: Suppose for some process q, $V_c[q] \neq \perp$ at the end of Phase 1. From Lemma 8, $V_c[q] = v_q$. Consider any $p \in \Pi_1$. We must show that $V_p[q] = v_q$ at the end of Phase 1. This is obvious if $p = c$, thus we consider the case where $p \neq c$.

Let r be the first round in which c received v_q (if $c = q$, we define r to be 0). From the algorithm, it is clear that $\Delta_c[q] = v_q$ at the end of round r. There are two cases to consider:

1. $r \leq n-2$. In round $r+1 \leq n-1$, c relays v_q by sending the message $(r+1, \Delta_c, c)$ with $\Delta_c[q] = v_q$ to all. From Lemma 10, p receives $(r+1, \Delta_c, c)$ in round $r+1$. From the algorithm, it is clear that p sets $V_p[q]$ to v_q by the end of round $r+1$.

2. $r = n-1$. In this case, c received v_q for the first time in round $n-1$. Since each process relays v_q (in its vector Δ) at most once, it is easy to see that v_q was relayed by all $n-1$ processes in $\Pi - \{c\}$, including p, before being received by c. Since p sets $V_p[q] = v_q$ before relaying v_q, it follows that $V_p[q] = v_q$ at the end of Phase 1.□

Lemma 12: For all $p \in \Pi_2$, $V_c = V_p$ at the end of Phase 2.

PROOF: Consider any $p \in \Pi_2$ and $q \in \Pi$. We have to show that $V_p[q] = V_c[q]$ at the end of Phase 2. There are two cases to consider:

1. $V_c[q] = v_q$ at the end of Phase 1. From Lemma 11, for all processes $p' \in \Pi_1$ (including p and c), $V_{p'}[q] = v_q$ at the end of Phase 1. Thus, for all the vectors V sent in Phase 2, $V[q] = v_q$. Hence, both $V_p[q]$ and $V_c[q]$ remain equal to v_q throughout Phase 2.

2. $V_c[q] = \perp$ at the end of Phase 1. Since $c \notin \mathcal{S}_p$, p waits for and receives V_c in Phase 2. Since $V_c[q] = \perp$, p sets $V_p[q] \leftarrow \perp$ at the end of Phase 2. □

Lemma 13: For all $p \in \Pi_2$, $V_p[c] = v_c$ at the end of Phase 2.

PROOF: It is clear from the algorithm that $V_c[c] = v_c$ at the end of Phase 1. From Lemma 11, for all $q \in \Pi_1$, $V_q[c] = v_c$ at the end of Phase 1. Thus, no process sends V with $V[c] = \perp$ in Phase 2. From the algorithm, it is clear that for all $p \in \Pi_2$, $V_p[c] = v_c$ at the end of Phase 2. □

Theorem 14: Given any Strong failure detector $\mathcal{S}$, the algorithm in Figure 5 solves Consensus in asynchronous systems with $f < n$.

PROOF: From the algorithm in Figure 5, it is clear that no process decides more than once, and this satisfies the uniform integrity requirement of Consensus. From Lemma 9,

every correct process eventually reaches Phase 3. From Lemma 13, the vector V_p of every correct has at least one non-$\perp$ component in Phase 3 (namely, $V_p[c] = v_c$). From the algorithm, every process p that reaches Phase 3, decides on the first non-$\perp$ component of V_p. Thus, every correct process decides some non-$\perp$ value in Phase 3—and this satisfies termination of Consensus. From Lemma 12, all processes that reach Phase 3 have the same vector V. Thus, all correct processes decide the same value, and agreement of Consensus is satisfied. From Lemma 8, this non-$\perp$ decision value is the proposed value of some process. Thus, uniform validity of Consensus is also satisfied. □

By Theorems 5 and 14, we have:

Corollary 15: Given any Weak failure detector $\mathcal{W}$, Consensus is solvable in asynchronous systems with $f < n$.

6.2 Using an Eventually Strong failure detector $\diamond\mathcal{S}$

Our previous solution to Consensus used $\mathcal{S}$, a failure detector that satisfies weak accuracy: at least one correct process was *never* suspected. We now solve Consensus using $\diamond\mathcal{S}$, a failure detector that only satisfies eventual weak accuracy. With $\diamond\mathcal{S}$, *all* processes may be erroneously added to the lists of suspects at one time or another. However, there is a correct process and a time after which that process is not suspected to have crashed. Note that at any given time t, processes cannot use $\diamond\mathcal{S}$ to determine whether any specific process is correct, or whether some correct process will never be suspected after time t.

Given any Eventually Strong failure detector $\diamond\mathcal{S}$, the algorithm in Figure 6 solves Consensus in asynchronous systems with a majority of correct processes. We show that solving Consensus using $\diamond\mathcal{S}$ requires this majority.[14] Thus, our algorithm is optimal with respect to the number of failures that it tolerates.

The algorithm in Figure 6 uses the *rotating coordinator* paradigm [Rei82, CM84, DLS88, BGP89, CT90]. Computation proceeds in asynchronous "rounds". We assume that all processes have a priori knowledge that during round r, the coordinator is process $c = (r \bmod n) + 1$. All messages are either to or from the "current" coordinator. Every time a process becomes a coordinator, it tries to determine a consistent decision value. If the current coordinator is correct *and* is not suspected by any surviving process, then it will succeed, and it will R-broadcast this decision value.

The algorithm in Figure 6 goes through three asynchronous epochs, each of which may span several asynchronous rounds. In the first epoch, several decision values are possible. In the second epoch, a value gets *locked*: no other decision value is possible. In the third epoch, processes decide the locked value.

Each round of this Consensus algorithm is divided into four asynchronous phases. In Phase 1, every process sends its current estimate of the decision value timestamped

[14] **In fact, we show that a majority of correct processes is required even if one uses $\diamond\mathcal{P}$, a stronger failure detector.**

Every process p executes the following:

procedure *propose*(v_p)
  $estimate_p \leftarrow v_p$ {*denotes p's estimate of the decision value*}
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$ {r_p *denotes the current round number*}
  $ts_p \leftarrow 0$ {*the round in which* $estimate_p$ *was last updated, initially 0*}

  {*Rotate through coordinators until decision is reached*}

  while $state_p = undecided$
    $r_p \leftarrow r_p + 1$
    $c_p \leftarrow (r_p \bmod n) + 1$ {c_p *is the current coordinator*}

    Phase 1: {*All processes p send* $estimate_p$ *to the current coordinator*}
      send $(p, r_p, estimate_p, ts_p)$ to c_p

    Phase 2: {*The current coordinator gathers* $n - f$ *estimates and proposes a new estimate*}
      if $p = c_p$ **then**
        wait until [for $n - f$ processes q : received $(q, r_p, estimate_q, ts_q)$ from q]
        $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received $(q, r_p, estimate_q, ts_q)$ from $q\}$
        $t \leftarrow$ largest ts_q such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
        $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t) \in msgs_p[r_p]$
        send $(p, r_p, estimate_p)$ to all

    Phase 3: {*All processes wait for the new estimate proposed by the current coordinator*}
      wait until [received $(c_p, r_p, estimate_{c_p})$ from c_p **or** $c_p \in \diamond\mathcal{S}_p$] {*Query the failure detector*}
      if [received $(c_p, r_p, estimate_{c_p})$ from c_p] **then** {p *received* $estimate_{c_p}$ *from* c_p}
        $estimate_p \leftarrow estimate_{c_p}$
        $ts_p \leftarrow r_p$
        send (p, r_p, ack) to c_p
      else send $(p, r_p, nack)$ to c_p {p *suspects that* c_p *crashed*}

    Phase 4: { *The current coordinator waits for* $n - f$ *replies. If these replies indicate that* $n - f$ *processes adopted its estimate, the coordinator sends a request to decide.* }
      if $p = c_p$ **then**
        wait until [for $n - f$ processes q : received (q, r_p, ack) **or** $(q, r_p, nack)$]
        if [for $n - f$ processes q : received (q, r_p, ack)] **then**
          R-broadcast$(p, r_p, estimate_p, decide)$

{*When p receives a decide message, it decides*}

when *R-deliver*$(q, r_q, estimate_q, decide)$
  if $state_p = undecided$ **then**
    decide$(estimate_q)$
    $state_p \leftarrow decided$

Figure 6: Solving Consensus using $\diamond S$

with the round number in which it adopted this estimate, to the current coordinator, c. In Phase 2, c gathers $n - f$ such estimates, selects one with the largest timestamp, and sends it to all the processes as their new estimate, $estimate_c$. In Phase 3, for each process p there are two possibilities:

1. p receives $estimate_c$ from c and sends an *ack* to c to indicate that it adopted $estimate_c$ as its own estimate; or

2. upon consulting its failure detector module $\diamond\mathcal{S}_p$, p *suspects* that c crashed, and sends a *nack* to c.

In Phase 4, c waits for $n - f$ replies (*ack*s or *nack*s). If all $n - f$ replies are *ack*s, then c knows that $n - f$ processes changed their estimates to $estimate_c$, and thus $estimate_c$ is locked. Consequently, c R-broadcasts a request to decide $estimate_c$. At any time, if a process R-delivers such a request, it decides accordingly.

The proof that the algorithm in Figure 6 solves Consensus is as follows. Let R be any run of the algorithm in Figure 6 using $\diamond\mathcal{S}$ in which all correct processes propose a value. We have to show that termination, uniform validity, agreement and uniform integrity hold.

Lemma 17: No two processes decide differently.[15]

PROOF: If no process ever decides, the lemma is trivially true. If any process decides, it must be the case that a coordinator R-broadcast a message of the type $(-, -, -, decide)$. This coordinator must have received $n - f$ messages of the type $(-, -, ack)$ in Phase 4. Let r be the smallest round number in which $n - f$ messages of the type $(-, r, ack)$ are sent to a coordinator in Phase 3. Let c denote the coordinator of round r, i.e., $c = (r \bmod n) + 1$. Let $estimate_c$ denote c's estimate at the end of Phase 2 of round r. We claim that for all rounds $r' \geq r$, if a coordinator c' sends $estimate_{c'}$ in Phase 2 of round r', then $estimate_{c'} = estimate_c$.

The proof is by induction on the round number. The claim trivially holds for $r' = r$. Now assume that the claim holds for all r', $r \leq r' < k$. Let c_k be the coordinator of round k, i.e., $c_k = (k \bmod n) + 1$. We will show that the claim holds for $r' = k$, i.e., if c_k sends $estimate_{c_k}$ in Phase 2 of round k, then $estimate_{c_k} = estimate_c$.

From the algorithm it is clear that if c_k sends $estimate_{c_k}$ in Phase 2 of round k then it must have received estimates from at least $n - f$ processes. Since $f < \frac{n}{2}$, there is some process p such that p sent a (p, r, ack) message to c in Phase 3 of round r and such that $(p, k, estimate_p, ts_p)$ is in $msgs_{c_k}[k]$ in Phase 2 of round k. Since p sent (p, r, ack) to c in Phase 3 of round r, $ts_p = r$ at the end of Phase 3 of round r. Since ts_p is non-decreasing, $ts_p \geq r$ in Phase 1 of round k. Thus in Phase 2 of round k, $(p, k, estimate_p, ts_p)$ is in $msgs_{c_k}[k]$ with $ts_p \geq r$. It is easy to see that there is no message $(q, k, estimate_q, ts_q)$ in

[15] This property, called *uniform agreement*, is stronger than the agreement requirement of Consensus which applies only to *correct* processes.

$msgs_{c_k}[k]$ for which $ts_q \geq k$. Let t be the largest ts_q such that $(q, k, estimate_q, ts_q)$ is in $msgs_{c_k}[k]$. Thus $r \leq t < k$.

In Phase 2 of round k, c_k executes $estimate_{c_k} \leftarrow estimate_q$ where $(q, k, estimate_q, t)$ is in $msgs_{c_k}[k]$. From Figure 6, it is clear that q adopted $estimate_q$ as its estimate in Phase 3 of round t. Thus, the coordinator of round t sent $estimate_q$ to q in Phase 2 of round t. Since $r \leq t < k$, by the induction hypothesis, $estimate_q = estimate_c$. Thus, c_k sets $estimate_{c_k} \leftarrow estimate_c$ in Phase 2 of round k. This concludes the proof of the claim.

We now show that if a process decides a value, then it decides $estimate_c$. Suppose that some process p R-delivers $(q, r_q, estimate_q, decide)$, and thus decides $estimate_q$. Process q must have R-broadcast $(q, r_q, estimate_q, decide)$ in Phase 4 of round r_q. From Figure 6, q must have received $n - f$ messages of the type $(-, r_q, ack)$ in Phase 4 of round r_q. By the definition of r, $r \leq r_q$. From the above claim, $estimate_q = estimate_c$. □

Lemma 18: Every correct process eventually decides some value.

PROOF: There are two possible cases:

1. Some correct process decides. It must have R-delivered some message of the type $(-, -, -, decide)$. By the agreement property of Reliable Broadcast, all correct processes eventually R-deliver this message and decide.

2. No correct process decides. We claim that no correct process remains blocked forever at one of the **wait** statements. The proof is by contradiction. Let r be the smallest round number in which some correct process blocks forever at one of the **wait** statements. Thus, all correct processes reach the end of Phase 1 of round r: they all send a message of the type $(-, r, estimate, -)$ to the current coordinator $c = (r \bmod n) + 1$. Therefore at least $n - f$ such messages are sent to c. There are two cases to consider:

   (a) Eventually, c receives those messages and replies by sending $(c, r, estimate_c)$. Thus, c does not block forever at the **wait** statement in Phase 2.

   (b) c crashes.

   In the first case, every correct process receives $(c, r, estimate_c)$. In the second case, since $\diamond\mathcal{S}$ satisfies *strong completeness*, for every correct process p there is a time after which c is permanently suspected by p, i.e., $c \in \diamond\mathcal{S}_p$. Thus in either case, no correct process blocks at the second **wait** statement (Phase 3). So every correct process sends a message of the type $(-, r, ack)$ or $(-, r, nack)$ to c in Phase 3. Since there are $n - f$ correct processes, c cannot block at the **wait** statement of Phase 4. This shows that all correct processes complete round r—a contradiction that completes the proof of our claim.

   Since $\diamond\mathcal{S}$ satisfies eventual weak accuracy, there is a correct process q and a time t such that no correct process suspects q after t. Thus, all processes that suspect

q after time t eventually crash and there is a time t' after which no process sends a message of the type $(-, r, nack)$ where q is the coordinator of round r (i.e., $q = (r \bmod n) + 1$). From this and the above claim, there must be a round r such that:

(a) All correct processes reach round r after time t' (when no process suspects q).

(b) q is the coordinator of round r (i.e., $q = (r \bmod n) + 1$).

In Phase 1 of round r, all correct processes send their estimates to q. In Phase 2, q receives $n - f$ such estimates, and sends $(q, r, estimate_q)$ to all processes. In Phase 3, since q is not suspected by any correct process after time t, every correct process waits for q's estimate, eventually receives it, and replies with an *ack* to q. Furthermore, no process sends a *nack* to q (that can only happen when a process suspects q). Thus in Phase 4, q receives $n - f$ messages of the type $(-, r, ack)$ (and no messages of the type $(-, r, nack)$), and q R-broadcasts $(q, r, estimate_q, decide)$. By the validity property of Reliable Broadcast, eventually all correct processes R-deliver q's message and *decide*—a contradiction. Thus case 2 is impossible, and this concludes the proof of the lemma. □

Theorem 19: Given any Eventually Strong failure detector $\diamond\mathcal{S}$, the algorithm in Figure 6 solves Consensus in asynchronous systems with $f < \frac{n}{2}$.

PROOF:

Termination: by Lemma 18.

Agreement: by Lemma 17.

Uniform integrity: It is clear from the algorithm that no process decides more than once.

Uniform validity: from the algorithm, it is clear that all the *estimates* that a coordinator receives in Phase 2 are proposed values. Therefore, the decision value that a coordinator selects from these *estimates* must be the value proposed by some process. Thus, uniform validity is satisfied. □

By Theorems 5 and 19, we have:

Corollary 20: Given any Eventually Weak failure detector $\diamond\mathcal{W}$, Consensus is solvable in asynchronous systems with $f < \frac{n}{2}$.

Thus, the weakest failure detector considered in this paper, $\diamond\mathcal{W}$, is sufficient to solve Consensus in asynchronous systems. This leads to the following question: What is the weakest failure detector for solving Consensus? Using the concept of reducibility, in [CHT92] we show that $\diamond\mathcal{W}$ is indeed the weakest failure detector for solving Consensus in asynchronous systems with a majority of correct processes. More precisely, we show:

Theorem 21: [CHT92] If a failure detector $\mathcal{D}$ can be used to solve Consensus in an asynchronous system, then $\mathcal{D} \succeq \diamond\mathcal{W}$ in that system.

By Corollary 20 and Theorem 21, we have:

Corollary 22: $\diamond\mathcal{W}$ is the weakest failure detector for solving Consensus in an asynchronous system with $f < \frac{n}{2}$.

6.3 A lower bound on fault-tolerance

In Section 6.1, we showed that failure detectors with *perpetual* accuracy (i.e., $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, or $\mathcal{W}$) can be used to solve Consensus in asynchronous systems with *any* number of failures. In contrast, with failure detectors with *eventual* accuracy (i.e., $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, or $\diamond\mathcal{W}$), our Consensus algorithms required a majority of the processes to be correct. We now show that this requirement is necessary: Any algorithm that uses $\diamond\mathcal{P}$ (the strongest of our four failure detectors with eventual accuracy) to solve Consensus requires a majority of correct processes. Thus, the algorithm in Figure 6 is optimal with respect to fault-tolerance.

Theorem 23: There is an Eventually Perfect failure detector $\diamond\mathcal{P}$ such that there is no algorithm A which solves Consensus using $\diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$.

PROOF: We now describe the behaviour of an Eventually Perfect failure detector $\diamond\mathcal{P}$ such that with every algorithm A, there is a run R_A of A using $\diamond\mathcal{P}$ that does not satisfy the specification of Consensus. Partition the processes into two sets Π_0 and Π_1 such that Π_0 contains $\lceil \frac{n}{2} \rceil$ processes, and Π_1 contains the remaining $\lfloor \frac{n}{2} \rfloor$ processes. Consider any Consensus algorithm A, and the following two runs of A using $\diamond\mathcal{P}$:

- Run $R_0 = \langle F_0, H_0, I, S_0, T_0 \rangle$: All processes in Π_0 propose 0, and all processes in Π_1 propose 1. All processes in Π_0 are correct in F_0, while those in Π_1 crash in F_0 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_0(t) = \Pi_1$ (this is possible since $f \geq \lceil \frac{n}{2} \rceil$). Every process in Π_0 permanently suspects every process in Π_1, i.e., $\forall t \in \mathcal{T}$, $\forall p \in \Pi_0 : H_0(p,t) = \Pi_1$. In this run, it is clear that $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect failure detector.

- Run $R_1 = \langle F_1, H_1, I, S_1, T_1 \rangle$: As in R_0, all processes in Π_0 propose 0, and all processes in Π_1 propose 1. All processes in Π_1 are correct in F_1, while those in Π_0 crash in F_1 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_1(t) = \Pi_0$. Every process in Π_1 permanently suspects every process in Π_0, i.e., $\forall t \in \mathcal{T}$, $\forall p \in \Pi_1 : H_1(p,t) = \Pi_0$. Clearly, $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect failure detector in this run.

Assume, without loss of generality, that both R_0 and R_1 satisfy the specifications of Consensus. Let $q_0 \in \Pi_0$, $q_1 \in \Pi_1$, t_0 be the time at which q_0 decides in R_0, and t_1 be the

time at which q_1 decides in R_1. There are three possible cases—in each case we construct a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of algorithm A using $\diamond\mathcal{P}$ such that $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect failure detector, but R_A violates the specification of Consensus.

1. In R_0, q_0 decides 1. Let $R_A = \langle F_0, H_0, I_A, S_0, T_0 \rangle$ be a run identical to R_0 except that all processes in Π_1 propose 0. Since in F_0 the processes in Π_1 crash right from the beginning of the run, R_0 and R_A are indistinguishable to q_0. Thus, q_0 decides 1 in R_A (as it did in R_0), thereby violating the uniform validity condition of Consensus.

2. In R_1, q_1 decides 0. This case is symmetric to Case 1.

3. In R_0, q_0 decides 0, and in R_1, q_1 decides 1. Construct $R_A = \langle F_A, H_A, I, S_A, T_A \rangle$ as follows. No processes crash in F_A, i.e., $\forall t \in \mathcal{T} : F_A(t) = \emptyset$. As before, all processes in Π_0 propose 0 and all processes in Π_1 propose 1. All messages from processes in Π_0 to those in Π_1 and vice-versa, are delayed until time $\max(t_0, t_1)$. Until time $\max(t_0, t_1)$, every process in Π_0 suspects every process in Π_1, and every process in Π_1 suspects every process in Π_0. After time $\max(t_0, t_1)$, no process suspects any other process, i.e.:

$$\begin{aligned}
&\forall t \leq \max(t_0, t_1) : \\
&\qquad \forall p \in \Pi_0 : H_A(p, t) = \Pi_1 \\
&\qquad \forall p \in \Pi_1 : H_A(p, t) = \Pi_0 \\
&\forall t > \max(t_0, t_1), \forall p \in \Pi : H_A(p, t) = \emptyset
\end{aligned}$$

   Clearly, $\diamond\mathcal{P}$ satisfies the specification of an Eventually Perfect failure detector.

   Until time $\max(t_0, t_1)$, R_A is indistinguishable from R_0 for processes in Π_0, and R_A is indistinguishable from R_1 for processes in Π_1. Thus in run R_A, q_0 decides 0 at time t_0, while q_1 decides 1 at time t_1. So q_0 and q_1 decide differently in R_A, and this violates the agreement condition of Consensus. □

In the Appendix, we refine the result of Theorem 23, by considering an infinite hierarchy of failure detectors ordered by the number of mistakes they can make, and showing exactly where in this hierarchy the majority requirement becomes necessary for solving Consensus (this hierarchy contains all eight failure detectors that we defined in Figure 1). Note that Theorem 23 is also a corollary of Theorem 4.3 in [DLS88] together with Theorem 35.

7 On Atomic Broadcast

We now consider Atomic Broadcast, another fundamental problem in fault tolerant distributed computing, and show that our results on Consensus also apply to Atomic Broadcast. Informally, Atomic Broadcast requires that all correct processes deliver the same

messages in the same order. Formally, *Atomic Broadcast* is a Reliable Broadcast that satisfies:

- *Total order*: If two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Total order and agreement ensure that all correct processes deliver the same *sequence* of messages. Atomic Broadcast is a powerful communication paradigm for fault-tolerant distributed computing [CM84, CASD85, BJ87, PGM89, BGT90, GSTC90, Sch90]. We now show that Consensus and Atomic Broadcast are *equivalent* in asynchronous systems with crash failures. This is shown by reducing each to the other.[16] In other words, a solution for one automatically yields a solution for the other. Both reductions apply to any asynchronous system (in particular, they do *not* require the assumption of a failure detector). This equivalence has important consequences regarding the solvability of Atomic Broadcast in *asynchronous* systems:

1. Atomic Broadcast can*not* be solved with a deterministic algorithm in asynchronous systems, even if we assume that at most one process may fail, and it can only fail by crashing. This is because Consensus has no deterministic solution in such systems [FLP85].

2. Atomic Broadcast can be solved using *randomization* or *unreliable failure detectors* in asynchronous systems. This is because Consensus is solvable with these techniques in such systems (for a survey of randomized Consensus algorithms, see [CD89]).

Consensus can be easily reduced to Atomic Broadcast as follows. To propose a value, a process atomically broadcasts it. To decide a value, a process picks the value of the first message that it atomically delivers.[17] By total order of Atomic Broadcast, all correct processes deliver the same first message. Hence they choose the same value and agreement of Consensus is satisfied. The other properties of Consensus are also easy to verify. In the next section, we reduce Atomic Broadcast to Consensus.

7.1 Reducing Atomic Broadcast to Consensus

In Figure 7, we show how to transform any Consensus algorithm into an Atomic Broadcast algorithm in asynchronous systems. The resulting Atomic Broadcast algorithm tolerates as many faulty processes as the given Consensus algorithm.

The reduction uses Reliable Broadcast, and repeated (possibly concurrent, but completely *independent*) executions of Consensus. Processes disambiguate between these

[16] They are actually equivalent even in asynchronous systems with arbitrary failures. However, the reduction is more complex and is omitted here.

[17] Note that this reduction does *not* require the assumption of a failure detector.

executions by tagging all the messages pertaining to the k^{th} execution of Consensus with the number k. Tagging each message with such a number constitutes a minor modification to any given Consensus algorithm. Informally, the k^{th} execution of Consensus is used to decide on the k^{th} batch of messages to be atomically delivered. The propose and decide primitives corresponding to the k^{th} execution of Consensus are denoted by *propose*$(k, -)$ and *decide*$(k, -)$.

Our Atomic Broadcast algorithm uses the *R-broadcast*(m) and *R-deliver*(m) primitives of Reliable Broadcast. To avoid possible ambiguities between Atomic Broadcast and Reliable Broadcast, we say that a process *A-broadcasts* or *A-delivers* to refer to a broadcast or a delivery associated with Atomic Broadcast; and *R-broadcasts* or *R-delivers* to refer to a broadcast or delivery associated with Reliable Broadcast.

When a process intends to A-broadcast a message m, it *R-broadcasts* m (in Task 1). When a process p R-delivers m, it adds m to the set $R\_delivered_p$ (Task 2). Thus, $R\_delivered_p$ contains all the messages submitted for Atomic Broadcast (since the beginning) that p is currently aware of. When p A-delivers a message m, it adds m to the set $A\_delivered_p$ (in Task 3). Thus, $R\_delivered_p - A\_delivered_p$ is the set of messages that were submitted for Atomic Broadcast but not yet A-delivered by p. This set is denoted by $A\_undelivered_p$. When $A\_undelivered_p$ is not empty, p proposes $A\_undelivered_p$ as the next batch of messages to be A-delivered. $batch_p(k)$ denotes the k^{th} batch of messages that p A-delivers: it is $msgSet_p$, the set of messages agreed upon by the k^{th} execution of Consensus, minus $A\_delivered_p$, those messages that p has already A-delivered.[18] Process p delivers the messages in $batch(k)$ in some deterministic order, e.g., lexicographical order, that was agreed *a priori* by all processes. This transformation of Consensus into Atomic Broadcast is described in Figure 7 as three concurrent and indivisible tasks. The proof of correctness follows.

Lemma 24: For any two correct processes p and q, and any message m, if $m \in R\_delivered_p$ then eventually $m \in R\_delivered_q$.

PROOF: If $m \in R\_delivered_p$ then p R-delivered m (in Task 2). Since p is correct, by agreement of Reliable Broadcast q eventually R-delivers m, and inserts m into $R\_delivered_q$. □

Lemma 25: For any two correct processes p and q, and all $k \geq 1$:

1. If p executes *propose*$(k, -)$, then q eventually executes *propose*$(k, -)$.
2. If p A-delivers messages in $batch_p(k)$, then q eventually A-delivers messages in $batch_q(k)$, and $batch_p(k) = batch_q(k)$.

PROOF: The proof is by simultaneous induction on (1) and (2). For $k = 1$, we first show that if p executes *propose*$(1, -)$, then q eventually executes *propose*$(1, -)$. When p

[18] It is possible for a process p to A-deliver a message m before it R-delivers m. This occurs if m was proposed by another process, and agreed upon by Consensus, before p R-delivers m.

Every process p executes the following:

Initialization:

$R\_delivered \leftarrow \emptyset$
$A\_delivered \leftarrow \emptyset$
$k \leftarrow 0$

To execute *A-broadcast*(m): { *Task 1* }

R-broadcast(m)

A-deliver$(-)$ occurs as follows:

when *R-deliver*(m) { *Task 2* }
$R\_delivered \leftarrow R\_delivered \cup \{m\}$

when $R\_delivered - A\_delivered \neq \emptyset$ { *Task 3* }
$k \leftarrow k + 1$
$A\_undelivered \leftarrow R\_delivered - A\_delivered$
propose$(k, A\_undelivered)$
wait until *decide*$(k, msgSet)$
$batch(k) \leftarrow msgSet - A\_delivered$
atomically deliver all messages in $batch(k)$ in some deterministic order
$A\_delivered \leftarrow A\_delivered \cup batch(k)$

Figure 7: Using Consensus to solve Atomic Broadcast

executes $propose(1, -)$, $R\_delivered_p$ must contain some message m. By Lemma 24, m is eventually in $R\_delivered_q$. Since $A\_delivered_q$ is initially empty, eventually $R\_delivered_q - A\_delivered_q \neq \emptyset$. Thus, q eventually executes Task 3 and $propose(1, -)$.

We now show that if p A-delivers messages in $batch_p(1)$, then q eventually A-delivers messages in $batch_q(1)$, and $batch_p(1) = batch_q(1)$. From the algorithm, if p A-delivers messages in $batch_p(1)$, it previously executed $propose(1, -)$. From part (1) of the lemma, all correct processes eventually execute $propose(1, -)$. By termination and uniform integrity of Consensus, every correct process eventually executes $decide(1, -)$ and it does so exactly once. By agreement of Consensus, all correct processes eventually execute $decide(1, msgSet)$ with the same $msgSet$. Since $A\_delivered_p$ and $A\_delivered_q$ are initially empty, $batch_p(1) = batch_q(1) = msgSet_p = msgSet_q$.

Now assume that the lemma holds for all k, $1 \leq k < l$. We first show that if p executes $propose(l, -)$, then q eventually executes $propose(l, -)$. When p executes $propose(l, -)$, $R\_delivered_p$ must contain some message m that is not in $A\_delivered_p$. Thus, m is not in $\bigcup_{k=1}^{l-1} batch_p(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. So m is not in $\bigcup_{k=1}^{l-1} batch_q(k)$. Since m is in $R\_delivered_p$, by Lemma 24, m is eventually in $R\_delivered_q$. Thus, there is a time after q A-delivers $batch_q(l-1)$ such that there is a message in $R\_delivered_q - A\_delivered_q$. So q eventually executes Task 3 and $propose(l, -)$.

We now show that if p A-delivers messages in $batch_p(l)$, then q A-delivers messages in $batch_q(l)$, and $batch_p(l) = batch_q(l)$. Since p A-delivers messages in $batch_p(l)$, it must have executed $propose(l, -)$. By part (1) of this lemma, all correct processes eventually execute $propose(l, -)$. By termination and uniform integrity of Consensus, every correct process eventually executes $decide(l, -)$ and it does so exactly once. By agreement of Consensus, all correct processes eventually execute $decide(l, msgSet)$ with the same $msgSet$. Note that $batch_p(l) = msgSet - \bigcup_{k=1}^{l-1} batch_p(k)$, and $batch_q(l) = msgSet - \bigcup_{k=1}^{l-1} batch_q(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. Thus, $batch_p(l) = batch_q(l)$. □

Lemma 26: The algorithm in Figure 7 satisfies agreement and total order.

PROOF: Immediate from Lemma 25, and the fact that correct processes A-deliver messages in each batch in the same deterministic order. □

Lemma 27: (Validity) If a correct process A-broadcasts m, then all correct processes eventually A-deliver m.

PROOF: The proof is by contradiction. Suppose some correct process p A-broadcasts m, and some correct process never A-delivers m. By Lemma 26, no correct process A-delivers m.

By Task 1 of Figure 7, p R-broadcasts m. By validity of Reliable Broadcast, every correct process q eventually R-delivers m, and inserts m in $R\_delivered_q$ (Task 2). Since correct processes never A-deliver m, they never insert m in $A\_delivered$. Thus, for every correct process q, there is a time after which m is *permanently* in $R\_delivered_q$ –

$A\_delivered_q$. From Figure 7 and Lemma 25, there is a k_1, such that for all $l > k_1$, all correct processes execute *propose*$(l, -)$, and they do so with sets that always include m.

Since all faulty processes eventually crash, there is a k_2 such that no faulty process executes *propose*$(l, -)$ with $l > k_2$. Let $k = max(k_1, k_2)$. Since all correct processes execute *propose*$(k, -)$, by termination and agreement of Consensus, all correct processes execute *decide*$(k, msgSet)$ with the same $msgSet$. By uniform validity of Consensus, some process q executed *propose*$(k, msgSet)$. From our definition of k, q is correct and $msgSet$ contains m. Thus all correct processes A-deliver m—a contradiction that concludes the proof. □

Lemma 28: (Uniform integrity) For any message m, each process A-delivers m at most once, and only if m was A-broadcast by some process.

PROOF: Suppose a process p A-delivers m. After p A-delivers m, it inserts m in $A\_delivered_p$. From the algorithm, it is clear that p cannot A-deliver m again.

From the algorithm, p executed *decide*$(k, msgSet)$ for some k and some $msgSet$ that contains m. By uniform validity of Consensus, some process q must have executed *propose*$(k, msgSet)$. So q previously R-delivered all the messages in $msgSet$, including m. By uniform integrity of Reliable Broadcast, some process r R-broadcast m. So, r A-broadcast m. □

Theorem 29: Consider any system (synchronous or asynchronous) subject to crash failures and where Reliable Broadcast can be implemented. The algorithm in Figure 7 transforms any algorithm for Consensus into an Atomic Broadcast algorithm.

PROOF: Immediate from Lemmata 26, 27, and 28. □

Since Reliable Broadcast can be implemented in asynchronous systems with crash failures (Section 4), the above theorem shows that Atomic Broadcast is reducible to Consensus in those systems. As we argued earlier, the converse is also true. Thus:

Corollary 30: Consensus and Atomic Broadcast are equivalent in asynchronous systems with crash failures.

The equivalence of Consensus and Atomic Broadcast in asynchronous systems immediately implies that our results regarding Consensus (in particular Corollaries 15 and 22, and Theorem 23) also hold for Atomic Broadcast:

Corollary 31: Given any Weak failure detector $\mathcal{W}$, Atomic Broadcast is solvable in asynchronous systems with $f < n$.

Corollary 32: $\diamond\mathcal{W}$ is the weakest failure detector for solving Atomic Broadcast in an asynchronous system with $f < \frac{n}{2}$.

Corollary 33: There is an Eventually Perfect failure detector $\diamond\mathcal{P}$ such that there is

no algorithm A which solves Atomic Broadcast using $\diamond\mathcal{P}$ in asynchronous systems with $f \geq \lceil \frac{n}{2} \rceil$.

Furthermore, Theorem 29 shows that by "plugging in" any *randomized* Consensus algorithm (such as the ones in [CD89]) into the algorithm of Figure 7, we automatically get a randomized algorithm for Atomic Broadcast in asynchronous systems.

Corollary 34: Atomic Broadcast can be solved by randomized algorithms in asynchronous systems with $f < \frac{n}{2}$ crash failures.

8 Related work

8.1 Partial synchrony

Fischer, Lynch and Paterson showed that Consensus cannot be solved in an asynchronous system subject to crash failures [FLP85]. The fundamental reason why Consensus cannot be solved in completely asynchronous systems is the fact that, in such systems, it is impossible to reliably distinguish a process that has crashed from one that is merely very slow. In other words, Consensus is unsolvable because accurate failure detection is impossible. On the other hand, it is well-known that Consensus is solvable (deterministically) in completely synchronous systems — that is, systems where clocks are perfectly synchronised, all processes take steps at the same rate and each message arrives at its destination a fixed and known amount of time after it is sent. In such a system we can use timeouts to implement a "perfect" failure detector — i.e., one in which no process is ever wrongly suspected, and every faulty process is eventually suspected. Thus, the ability to solve Consensus in a given system is intimately related to the failure detection capabilities of that system. This realisation led us to augment the asynchronous model of computation with unreliable failure detectors as described in this paper.

A different tack on circumventing the unsolvability of Consensus is pursued in [DDS87] and [DLS88]. The approach of those papers is based on the observation that between the completely synchronous and completely asynchronous models of distributed systems there lie a variety of intermediate "partially synchronous" models.

In particular, [DDS87] defines a space of 32 models by considering five key parameters, each of which admits a "favourable" and an "unfavourable" setting. For instance, one of the parameters is whether the maximum message delay is bounded and known (favourable setting) or unbounded (unfavourable setting). Each of the 32 models corresponds to a particular setting of the 5 parameters. [DDS87] identifies four "minimal" models in which Consensus is solvable. These are minimal in the sense that the weakening of any parameter from favourable to unfavourable would yield a model of partial synchrony where Consensus is unsolvable. Thus, within the space of the models considered, [DDS87] delineates precisely the boundary between solvability and unsolvability of Consensus, and provides an answer to the question "What is the least amount of

Every process p executes the following:

```
output_p ← ∅
for all q ∈ Π                    {Δ_p(q) denotes the duration of p's time-out interval for q}
    Δ_p(q) ← default time-out interval

cobegin
|| Task 1: repeat periodically
    send "p-is-alive" message to all

|| Task 2: repeat periodically
    for all q ∈ Π
        if q ∉ output_p and p did not receive "q-is-alive" in the last Δ_p(q) seconds
            output_p ← output_p ∪ {q}
                                  {p times-out on q: it now suspects q has crashed}

|| Task 3: when receive "q-is-alive" for some q
    if q ∈ output_p              {p knows that it prematurely timed-out on q:}
        output_p ← output_p − {q}      {1. p repents on q, and}
        Δ_p(q) ← Δ_p(q) + 1            {2. p increases its time-out period for q}
coend
```

Figure 8: A time-out based implementation of $\diamond\mathcal{P}$ in some models of partial synchrony.

synchrony sufficient to solve Consensus?".

[DLS88] considers the following two models of partial synchrony. The first model assumes that there are bounds on relative process speeds and on message transmission times, but these bounds are not known. The second model assumes that these bounds are known, but they hold only after some unknown time.

In each one of these two models (with crash failures), it is easy to implement an Eventually Perfect failure detector $\diamond\mathcal{P}$. In fact, we can implement $\diamond\mathcal{P}$ in an even weaker model of partial synchrony: one in which there are bounds on message transmission times and relative process speeds, but these bounds are not known *and* they hold only after some unknown time. Since $\diamond\mathcal{P}$ is stronger than $\diamond\mathcal{W}$, by Corollaries 20 and 32, this implementation immediately gives Consensus and Atomic Broadcast solutions for this model of partial synchrony and, a fortiori, for the two models of [DLS88]. The implementation of $\diamond\mathcal{P}$ is given in Figure 8, and proven below.

Each process p periodically sends a "p-is-alive" message to all the processes. If p does not receive a "q-is-alive" message from some process q for $\Delta_p(q)$ units of time, p adds q to its list of suspects. If p receives "q-is-alive" from some process q that it currently

suspects, p knows that its previous time-out on q was premature. In this case, p removes q from its list of suspects and increases the length of the time-out.

Theorem 35: Consider a system in which, after some time t, some bounds on relative process speeds and on message transmission times hold (we do not assume that t or the value of these bounds are known). The algorithm in Figure 8 implements an Eventually Perfect failure detector $\diamond\mathcal{P}$ in this system.

PROOF: (*sketch*) We first show that strong completeness holds, i.e., eventually every process that crashes is permanently suspected by every correct process. Suppose a process q crashes. Clearly, q eventually stops sending "q-is-alive" messages, and there is a time after which no correct process receives such a message. Thus, there is a time t' after which: (1) all correct processes time-out on q (Task 2), and (2) they do not receive any message from q after this time-out. From the algorithm, it is clear that after time t', all correct processes will permanently suspect q. Thus, strong completeness is satisfied.

We now show that eventual strong accuracy is satisfied. That is, for any correct processes p and q, there is a time after which p will not suspect q. There are two possible cases:

1. Process p times-out on q finitely often (in Task 2). Since q is correct and keeps sending "q-is-alive" messages forever, eventually p receives one such message after its last time-out on q. At this point, q is permanently removed from p's list of suspects (Task 3).

2. Process p times-out on q infinitely often (in Task 2). Note that p times-out on q (and so p adds q to $output_p$) only if q is not already in $output_p$. Thus, q is added to and removed from $output_p$ infinitely often. Process q is only removed from $output_p$ in Task 3, and every time this occurs the time-out period $\Delta_p(q)$ is increased. Since this occurs infinitely often, $\Delta_p(q)$ grows unbounded. Thus, eventually (1) the bounds on relative process speeds and on message transmission times hold, and (2) $\Delta_p(q)$ is larger than the *correct* time-out based on these bounds. After this point, p cannot time-out on q any more—a contradiction to our assumption that p times-out on q infinitely often. Thus Case 2 cannot occur. □

Thus, failure detectors can be viewed as a more abstract and modular way of incorporating partial synchrony assumptions into the model of computation. Instead of focusing on the *operational features* of partial synchrony (such as the five parameters considered in [DDS87]), we can consider the *axiomatic properties* that failure detectors must have in order to solve Consensus. The problem of implementing a given failure detector in a specific model of partial synchrony becomes a separate issue; this separation affords greater modularity.

Studying failure detectors rather than various models of partial synchrony has other advantages as well. By showing that Consensus is solvable using some specific failure detector we thereby show that Consensus is solvable in *all* systems in which that failure

detector can be implemented. An algorithm that relies on the axiomatic properties of a given failure detector is more general, more modular, and simpler to understand than one that relies directly on specific operational features of partial synchrony (that can be used to implement the given failure detector).

From this more abstract point of view, the question "What is the least amount of synchrony sufficient to solve Consensus?" translates to "What is the weakest failure detector sufficient to solve Consensus?". In contrast to [DDS87], which identified a *set* of minim*al* models of partial synchrony in which Consensus is solvable, in [CHT92] we are able to exhibit a *single* minim*um* failure detector, $\diamond\mathcal{W}$, that can be used to solve Consensus. The technical device that made this possible is the notion of *reduction* between failure detectors.

8.2 The application of failure detection in shared memory systems

Loui and Abu-Amara showed that in an asynchronous shared memory system with atomic read/write registers, Consensus cannot be solved even if at most one process may crash [LA87]. This raises the following natural question: can we circumvent this impossibility result using unreliable failure detectors? In a recent work, Lo shows that this is indeed possible [Lo93]. In particular, he shows that using a Strong failure detector and atomic registers, one can solve Consensus for any number of failures. He also shows that for systems with a majority of correct processes, it is sufficient to use an Eventually Strong failure detector and atomic registers.

8.3 The Isis toolkit

With our approach, even if a correct process p is repeatedly suspected to have crashed by the other processes, it is still required to behave like every other correct process in the system. For example, with Atomic Broadcast, p is still required to A-deliver the same messages, in the same order, as all the other correct processes. Furthermore, p is not prevented from A-broadcasting messages, and these messages must eventually be A-delivered by *all* correct processes (including those processes whose local failure detector modules permanently suspect p to have crashed). In summary, application programs that use unreliable failure detection are aware that the information they get from the failure detector may be incorrect: they only take this information as an imperfect "hint" about which processes have really crashed. Furthermore, processes are never "discriminated against" if they are falsely suspected to have crashed.

Isis takes an alternative approach based on the assumption that failure detectors rarely make mistakes [RB91]. In those cases in which a correct process p is falsely suspected by the failure detector, p is effectively forced "to crash" (via a *group membership* protocol that removes p from all the groups that it belongs to). An application using such a failure detector cannot distinguish between a faulty process that really crashed,

and a correct one that was forced to do so. Essentially, the Isis failure detector forces the system to conform to its view. From the application's point of view, this failure detector looks "perfect": it never makes visible mistakes.

For the Isis approach to work, the low-level time-outs used to detect crashes must be set very conservatively: Premature time-outs are costly (each results in the removal of a process), and too many of them can lead to system shutdown.[19] In contrast, with our approach, premature time-outs (e.g., failure detector mistakes) are not so deleterious: they can only *delay* an application. In other words, premature time-outs can affect the *liveness* but not the *safety* of an application. For example, consider the Atomic Broadcast algorithm that uses $\diamond\mathcal{W}$. If the failure detector "malfunctions", some messages may be delayed, but no message is ever delivered out of order, and no correct process is removed. If the failure detector stops malfunctioning, outstanding messages are eventually delivered. Thus, we can set time-out periods more aggressively than Isis: in practice, we would set our failure detector time-out periods closer to the average case, while Isis must set time-outs to the worst-case.

8.4 Other work

Several works in fault-tolerant computing used time-outs primarily or exclusively for the purpose of failure detection. An example of this approach is given by an algorithm in [ADLS91], which, as pointed out by the authors, "can be viewed as an asynchronous algorithm that uses a fault detection (e.g., *timeout*) mechanism."

Acknowledgements

We are deeply grateful to Vassos Hadzilacos for his help in revising this paper. We would like to thank Prasad Jayanti for greatly simplifying the algorithm in Figure 3. We would also like to thank Özalp Babaoğlu, Ken Birman, Navin Budhiraja, Dexter Kozen, Keith Marzullo, Gil Neiger, Anil Nerode, Aleta Ricciardi, Fred Schneider and the distributed systems group at Cornell for their comments and criticisms.

References

[ABD+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, October 1987.

[19] For example, the time-out period in the current version of Isis is greater than 10 seconds.

[ADKM91] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. Technical Report CS91-13, Computer Science Department, The Hebrew University of Jerusalem, November 1991.

[ADLS91] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainity. In *Proceedings of the Twenty third ACM Symposium on Theory of Computing*, May 1991.

[BCJ+90] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Kenneth P. Kane, and Frank Bernhard Schmuck. *ISIS - A Distributed Programming Environment*, June 1990.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus. In *Proceedings of the Thirtieth Symposium on Foundations of Computer Science*, pages 410–415. IEEE Computer Society Press, October 1989.

[BGT90] Navin Budhiraja, Ajei Gopal, and Sam Toueg. Early-stopping distributed bidding and applications. In *Proceedings of the Fourth International Workshop on Distributed Algorithms*. Springer-Verlag, September 1990. In press.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[BW87] M. Bridgland and R. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, 1987.

[CASD85] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CDD90] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. Technical Report RJ 7424, IBM Research Laboratory, April 1990.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Technical Report 92-1293, Department of Computer Science, Cornell University, July 1992. Available by anonymous ftp from `ftp.cs.cornell.edu` in `pub/chandra/failure.detectors.weakest.dvi.Z`. A preliminary version appeared in the *Proceedings of the Eleventh ACM Symposium on Principles of Distributed Computing*, pages 147–158. ACM press, August 1992.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Cri87] Flaviu Cristian. Issues in the design of highly available computing services. In *Annual Symposium of the Canadian Information Processing Society*, pages 9–16, July 1987. Also IBM Research Report RJ5856, July 1987.

[CT90] Tushar Deepak Chandra and Sam Toueg. Time and message efficient reliable broadcasts. In *Proceedings of the Fourth International Workshop on Distributed Algorithms*. Springer-Verlag, September 1990. In press.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Fis83] Michael J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). Technical Report 273, Department of Computer Science, Yale University, June 1983.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[GSTC90] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 297–310, August 1990.

[HM90] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, July 1990.

[LA87] M.C. Loui and Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in computing research*, 4:163–183, 1987.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Lo93] Wai Kau Lo. Using failure detectors to solve consensus in asynchronous shared-memory systems. Master's thesis, University of Toronto, January 1993.

[MDH86] Yoram Moses, Danny Dolev, and Joseph Y. Halpern. Cheating husbands and other stories: a case study of knowledge, action, and communication. *Distributed Computing*, 1(3):167–176, 1986.

[Mul87] Sape J. Mullender, editor. *The Amoeba distributed operating system: Selected papers 1984 - 1987.* Centre for Mathematics and Computer Science, 1987.

[PBS89] L. L. Peterson, N. C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. In *ACM Transactions on computer systems 7,3*, pages 217–246, August 1989.

[PGM89] Frank Pittelli and Hector Garcia-Molina. Reliable scheduling in a tmr database system. *ACM Transactions on Computer Systems*, 7(1):25–60, February 1989.

[Pow91] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing.* Springer-Verlag, 1991.

[RB91] Aleta Ricciardi and Ken Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351. ACM Press, August 1991.

[Rei82] Rüdiger Reischuk. A new solution for the Byzantine general's problem. Technical Report RJ 3673, IBM Research Laboratory, November 1982.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock.

SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.

Appendix: A hierarchy of failure detectors and bounds on fault-tolerance

In the preceding sections, we introduced the concept of unreliable failure detectors that could make mistakes, and showed how to use them to solve Consensus despite such mistakes. Informally, a mistake occurs when a correct process is erroneously added to the list of processes that are suspected to have crashed. In this Appendix, we formalise this concept and study a related property that we call *repentance*. Informally, if a process p learns that its failure detector module $\mathcal{D}_p$ made a mistake, repentance requires $\mathcal{D}_p$ to take corrective action. Based on mistakes and repentance, we define a hierarchy of failure detector specifications that will be used to unify some of our results, and to refine the lower bound on fault-tolerance given in Section 6.3. This infinite hierarchy consists of a continuum of repentant failure detectors ordered by the maximum number of mistakes that each one can make.

Mistakes and Repentance

We now define a *mistake*. Let $R = \langle F, H, I, S, T \rangle$ be any run using a failure detector $\mathcal{D}$. $\mathcal{D}$ makes a *mistake in R at time t on process p about process q* if at time t, p begins to suspect that q has crashed even though $q \notin F(t)$. Formally:

$$[q \notin F(t), q \in H(p,t)] \text{ and } [\exists t' < t, \forall t'', t' \leq t'' < t : q \notin H(p,t'')]$$

Such a mistake is denoted by the tuple $\langle R, p, q, t \rangle$. The set of mistakes made by $\mathcal{D}$ in R is denoted by $M(R)$.

Note that only the erroneous *addition* of q into $\mathcal{D}_p$ is counted as a mistake on p. The continuous *retention* of q into $\mathcal{D}_p$ does not count as additional mistakes. Thus, a failure detector can make multiple mistakes on a process p about another process q only by repeatedly adding and then removing q from the set $\mathcal{D}_p$. In practice, mistakes are caused by premature time-outs.

We define the following four types of accuracy properties for a failure detector $\mathcal{D}$ based on the mistakes made by $\mathcal{D}$:

- *Strongly $k-$mistaken:* $\mathcal{D}$ makes at most k mistakes. Formally, $\mathcal{D}$ is strongly $k-$mistaken if:

$$\forall R \text{ using } \mathcal{D} : |M(R)| \leq k$$

- *Weakly $k-$mistaken:* There is a correct process p such that $\mathcal{D}$ makes at most k mistakes about p. Formally, $\mathcal{D}$ is weakly $k-$mistaken if:

$$\forall R = \langle F, H, I, S, T \rangle \text{ using } \mathcal{D}, \exists p \in \mathit{correct}(F) : |\{\langle R, q, p, t \rangle : \langle R, q, p, t \rangle \in M(R)\}| \leq k$$

- *Strongly finitely mistaken:* $\mathcal{D}$ makes a finite number of mistakes. Formally, $\mathcal{D}$ is strongly finitely mistaken if:

$$\forall R \text{ using } \mathcal{D} : M(R) \text{ is finite.}$$

  In this case, it is clear that there is a time t after which $\mathcal{D}$ stops making mistakes.

- *Weakly finitely mistaken:* There is a correct process p such that $\mathcal{D}$ makes a finite number of mistakes about p. Formally, $\mathcal{D}$ is weakly finitely mistaken if:

$$\forall R = \langle F, H, I, S, T \rangle \text{ using } \mathcal{D}, \exists p \in \mathit{correct}(F) : \{\langle R, q, p, t \rangle : \langle R, q, p, t \rangle \in M(R)\} \text{ is finite.}$$

  In this case, there is a time t after which $\mathcal{D}$ stops making mistakes about p.

For most values of k, the properties mentioned above are not powerful enough to be useful. For example, suppose every process permanently suspects every other process. In this case, the failure detector makes at most $(n-1)^2$ mistakes, but it is clearly useless since it does not provide any information.

The core of this problem is that such failure detectors are not forced to reverse a mistake, even when a mistake becomes "obvious" (say, after a process q replies to an inquiry that was sent to q after q was suspected to have crashed). However, we can impose a natural requirement to circumvent this problem. Consider the following scenario. The failure detector module at process p erroneously adds q to $\mathcal{D}_p$ at time t. Subsequently, p sends a message to q and receives a reply. This reply is a proof that q had not crashed at time t. Thus, p *knows* that its failure detector module made a mistake about q. It is reasonable to require that, given such irrefutable evidence of a mistake, the failure detector module at p takes the corrective action of removing q from $\mathcal{D}_p$. In general, we can require the following property:

- *Repentance:* If a correct process p eventually *knows* that $q \notin F(t)$, then at some time after t, $q \notin \mathcal{D}_p$. Formally, $\mathcal{D}$ is *repentant* if:

$$\forall R = \langle F, H, I, S, T \rangle \text{ using } \mathcal{D}, \forall t, \forall p, q \in \Pi : [\exists t' : (R, t') \models K_p(q \notin F(t))] \Rightarrow [\exists t'' \geq t : q \notin H(p, t'')]$$

The knowledge theoretic operator K_p can be defined formally [HM90]. Informally $(R,t) \models \phi$ iff in run R at time t, predicate ϕ holds. We say $(R,t) \sim_p (R',t')$ iff the run R at time t and the run R' at time t' are indistinguishable to p. Finally, $(R,t) \models K_p(\phi) \iff [\forall (R',t') \sim_p (R,t) : (R',t') \models \phi]$. For a detailed treatment of Knowledge Theory as applied to distributed systems, the reader should refer to the seminal work done in [MDH86, HM90].

Recall that in Section 2.2 we defined a failure detector to be a function that maps each failure pattern to a set of failure detector histories. Thus, the specification of a failure detector depends solely on the failure pattern actually encountered. In contrast, the definition of repentance depends on the knowledge (about mistakes) at each process. This in turn depends on the algorithm being executed, and the communication pattern actually encountered. Thus, repentant failure detectors cannot be specified solely in terms of the failure pattern actually encountered. Nevertheless, repentance is an important property that we would like many failure detectors to satisfy.

In the rest of this Appendix, we informally define a hierarchy of repentant failure detectors that differ by their accuracy (i.e., the maximum number of mistakes they can make). As we just noted, such failure detectors cannot be specified solely in terms of the failure pattern actually encountered, and thus they do not fit the formal definition of failure detectors given in Section 2.2.

A hierarchy of repetant failure detectors

We now define an infinite hierarchy of repentant failure detectors. Every failure detector in this hierarchy satisfies weak completeness, repentance, and one of the four types of accuracy that we defined in the previous section. We name these failure detectors after the accuracy property that they satisfy:

- $\mathcal{SF}(k)$ denotes a *Strongly k-Mistaken failure detector*,
- $\mathcal{SF}$ denotes a *Strongly Finitely Mistaken failure detector*,
- $\mathcal{WF}(k)$ denotes a *Weakly k-Mistaken failure detector*, and
- $\mathcal{WF}$ denotes a *Weakly Finitely Mistaken failure detector*.

Clearly, $\mathcal{SF}(0) \succeq \mathcal{SF}(1) \succeq \ldots \mathcal{SF}(k) \succeq \mathcal{SF}(k+1) \succeq \ldots \succeq \mathcal{SF}$. A similar order holds for the $\mathcal{WF}$s. Consider a system of n processes of which at most f may crash. In this system, there are at least $n-f$ correct processes. Since $\mathcal{SF}((n-f)-1)$ makes fewer mistakes than the number of correct processes, there is at least one correct process that it never suspects. Thus, $\mathcal{SF}((n-f)-1)$ is weakly 0-mistaken, and $\mathcal{SF}((n-f)-1) \succeq \mathcal{WF}(0)$. Furthermore, it is clear that $\mathcal{SF} \succeq \mathcal{WF}$. This infinite hierarchy of failure detectors, ordered by reducibility, is illustrated in Figure 9 (where an edge $\rightarrow$ denotes the $\succeq$ relation).

Each of the eight failure detectors that we considered in Section 2.4 is equivalent to some failure detector in this hierarchy. In particular, it is easy to show that:

Observation 36:

- $\mathcal{P} \cong \mathcal{Q} \cong \mathcal{SF}(0)$,
- $\mathcal{S} \cong \mathcal{W} \cong \mathcal{WF}(0)$,

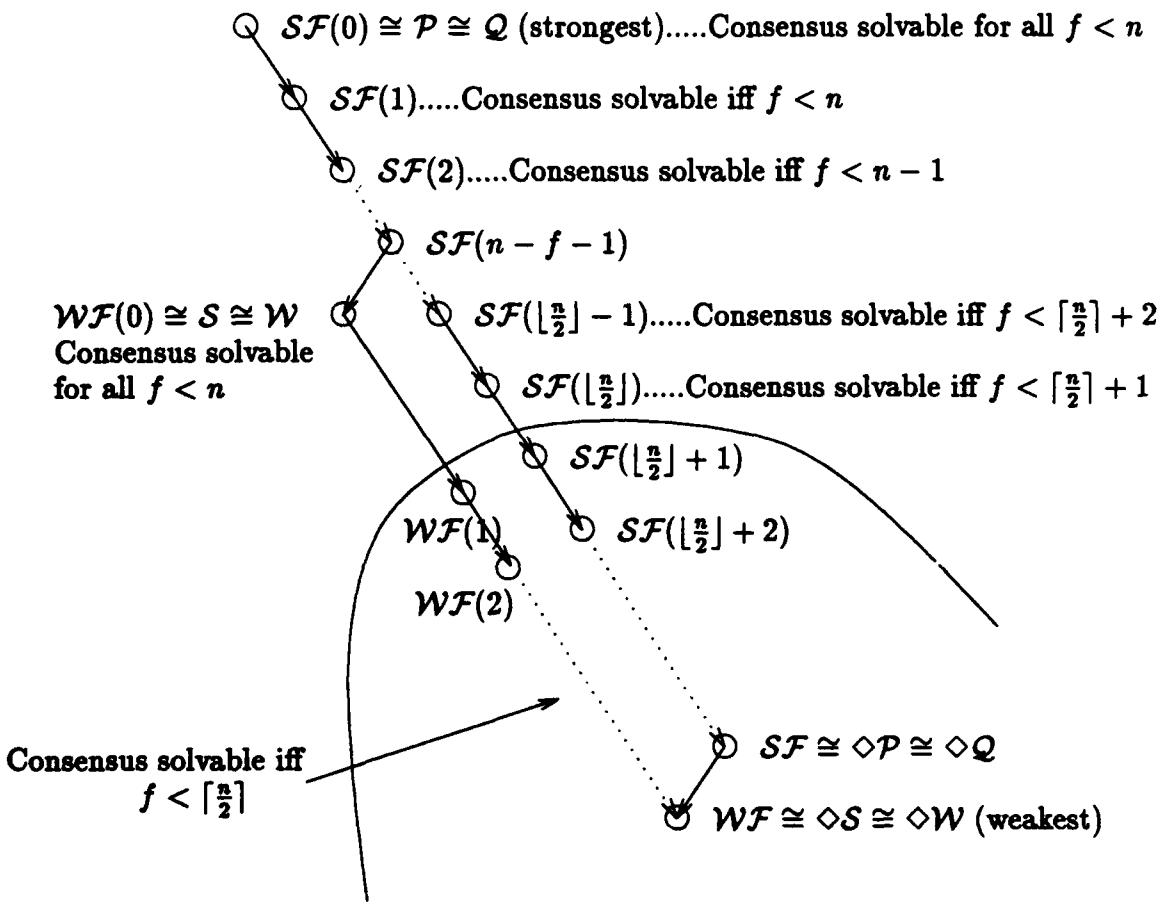


Figure 9: The hierarchy of repentant failure detectors ordered by reducibility. This figure also shows the maximum number of faulty processes for which Consensus can be solved using each failure detector in this hierarchy.

- $\diamond\mathcal{P} \cong \diamond\mathcal{Q} \cong \mathcal{SF}$, and
- $\diamond\mathcal{S} \cong \diamond\mathcal{W} \cong \mathcal{WF}$.

For example, it is easy to see that the reduction algorithm in Figure 3 transforms $\mathcal{WF}$ into $\diamond\mathcal{W}$. Other conversions are similar or straightforward and are therefore omitted. Note that $\mathcal{P}$ and $\diamond\mathcal{W}$ are the strongest and weakest failure detectors in this hierarchy, respectively. From Corollaries 15 and 31, and Observation 36 we have:

Corollary 37: Given $\mathcal{WF}(0)$, Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < n$.

Similarly, from Corollaries 20 and 32, and Observation 36 we have:

Corollary 38: Given $\mathcal{WF}$, Consensus and Atomic Broadcast are solvable in asynchronous systems with $f < \frac{n}{2}$.

Tight bounds on fault-tolerance

Since Consensus and Atomic Broadcast are equivalent in asynchronous systems with any number of faulty processes (Theorem 30), we can focus on establishing fault-tolerance bounds for Consensus. In Section 6, we showed that failure detectors with *perpetual* accuracy (i.e., $\mathcal{P}$, $\mathcal{Q}$, $\mathcal{S}$, or $\mathcal{W}$) can be used to solve Consensus in asynchronous systems with *any* number of failures. In contrast, with failure detectors with *eventual* accuracy (i.e., $\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, or $\diamond\mathcal{W}$), Consensus can be solved if and only if a majority of the processes are correct. We now refine this result by considering each failure detector $\mathcal{D}$ in our infinite hierarchy of failure detectors, and determining how many correct processes are necessary to solve Consensus using $\mathcal{D}$. The results are illustrated in Figure 9.

There are two cases depending on whether we assume that the system has a majority of correct processes or not. Since $\diamond\mathcal{W}$, the weakest failure detector in the hierarchy, can be used to solve Consensus when a majority of the processes are correct, we have:

Observation 39: If $f < \frac{n}{2}$ then Consensus can be solved using any failure detector in the hierarchy of Figure 9.

We now consider the solvability of Consensus in systems that do not have a majority of correct processes. For these systems, we determine the maximum m for which Consensus is solvable using $\mathcal{SF}(m)$ or $\mathcal{WF}(m)$. We first show that Consensus is solvable using $\mathcal{SF}(m)$ if and only if m, the number of mistakes, is less than or equal to $n - f$, the number of correct processes. We then show that Consensus is solvable using $\mathcal{WF}(m)$ if and only if $m = 0$.

Theorem 40: Suppose $f \geq \frac{n}{2}$. If $m > n - f$ then there is a Strongly m-Mistaken failure detector $\mathcal{SF}(m)$ such that there is no algorithm A which solves Consensus using $\mathcal{SF}(m)$ in asynchronous systems.

PROOF: [*sketch*] We describe the behaviour of a Strongly m-Mistaken failure detector $\mathcal{SF}(m)$ such that with every algorithm A, there is a run R_A of A using $\mathcal{SF}(m)$ that does not satisfy the specification of Consensus. Since $1 \leq n - f \leq \frac{n}{2}$, we can partition the processes into three sets Π_0, Π_1 and $\Pi_{crashed}$, such that Π_0 and Π_1 are non-empty sets containing $n - f$ processes each, and $\Pi_{crashed}$ is a (possibly empty) set containing the remaining $n - 2(n - f)$ processes. For the rest of this proof, we will only consider runs in which all processes in $\Pi_{crashed}$ crash in the beginning of the run. Let $q_0 \in \Pi_0$ and $q_1 \in \Pi_1$. Consider any Consensus algorithm A, and the following two runs of A using $\mathcal{SF}(m)$:

- Run $R_0 = \langle F_0, H_0, I, S_0, T_0 \rangle$: All processes in Π_0 propose 0, and all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1. All processes in Π_0 are correct in F_0, while all the f processes in $\Pi_1 \cup \Pi_{crashed}$ crash in F_0 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_0(t) = \Pi_1 \cup \Pi_{crashed}$. Process q_0 permanently suspects every process in $\Pi_1 \cup \Pi_{crashed}$, i.e., $\forall t \in \mathcal{T} : H_0(q_0, t) = \Pi_1 \cup \Pi_{crashed} = F_0(t)$. No other process suspects any process, i.e., $\forall t \in \mathcal{T}, \forall q \neq q_0 : H_0(q, t) = \emptyset$. In this run, it is clear that $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector.

- Run $R_1 = \langle F_1, H_1, I, S_1, T_1 \rangle$: As in R_0, all processes in Π_0 propose 0, and all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1. All processes in Π_1 are correct in F_1, while all the f processes in $\Pi_0 \cup \Pi_{crashed}$ crash in F_1 at the beginning of the run, i.e., $\forall t \in \mathcal{T} : F_1(t) = \Pi_0 \cup \Pi_{crashed}$. Process q_1 permanently suspects every process in $\Pi_0 \cup \Pi_{crashed}$, and no other process suspects any process. Clearly, $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector in this run.

Assume, without loss of generality, that both R_0 and R_1 satisfy the specification of Consensus. Let t_0 be the time at which q_0 decides in R_0, and let t_1 be the time at which q_1 decides in R_1. There are three possible cases—in each case we construct a run $R_A = \langle F_A, H_A, I_A, S_A, T_A \rangle$ of algorithm A using $\mathcal{SF}(m)$ such that $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector, but R_A violates the specification of Consensus.

1. In R_0, q_0 decides 1. Let $R_A = \langle F_0, H_0, I_A, S_0, T_0 \rangle$ be a run identical to R_0 except that all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 0. Since in F_0 the processes in $\Pi_1 \cup \Pi_{crashed}$ crash right from the beginning of the run, R_0 and R_A are indistinguishable to q_0. Thus, q_0 decides 1 in R_A (as it did in R_0), thereby violating the uniform validity condition of Consensus.

2. In R_1, q_1 decides 0. This case is symmetric to Case 1.

3. In R_0, q_0 decides 0, and in R_1, q_1 decides 1. Construct $R_A = \langle F_A, H_A, I, S_A, T_A \rangle$ as follows. As before, all processes in Π_0 propose 0, all processes in $\Pi_1 \cup \Pi_{crashed}$ propose 1, and all processes in $\Pi_{crashed}$ crash in F_A at the beginning of the run. All messages from processes in Π_0 to those in Π_1 and vice-versa, are delayed until time

$t_0 + t_1$. Until time t_0, R_A is identical to R_0, except that the processes in Π_1 do not crash, they are only "very slow" and do not take any steps before time t_0. Thus, until time t_0, q_0 cannot distinguish between R_0 and R_A, and it decides 0 at time t_0 in R_A (as it did in R_0). Note that by time t_0, the failure detector $\mathcal{SF}(m)$ made $n - f$ mistakes in R_A: q_0 erroneously suspected that all processes in Π_1 crashed (while they were only slow).

From time t_0, the construction of R_A continues as follows.

(a) At time t_0, all processes in Π_0, except q_0, crash in F_A.

(b) From time t_0 to time $t_0 + t_1$, q_1 suspects all processes in $\Pi_0 \cup \Pi_{crashed}$, i.e., $\forall t, t_0 \leq t \leq t_0 + t_1 : H_A(q_1, t) = \Pi_0 \cup \Pi_{crashed}$, and no other process suspects any process. By suspecting all the processes in Π_0, including q_0, the failure detector makes one mistake on process q_1 (about q_0). Thus, by time $t_0 + t_1$, $\mathcal{SF}(m)$ has made a total of $(n - f) + 1$ mistakes in R_A. Since $m > n - f$, $\mathcal{SF}(m)$ has made at most m mistakes in R_A until time $t_0 + t_1$.

(c) At time t_0, processes in Π_1 "wake up." From time t_0 to time $t_0 + t_1$ they execute exactly as they did in R_1 from time 0 to time t_1 (they cannot perceive this real-time shift of t_0). Thus, at time $t_0 + t_1$ in run R_A, q_1 decides 1 (as it did at time t_1 in R_1). So q_0 and q_1 decide differently in R_A, and this violates the agreement condition of Consensus.

(d) From time $t_0 + t_1$ onwards the run R_A continues as follows. No more processes crash and every correct process suspects exactly all the processes that have crashed. Thus, $\mathcal{SF}(m)$ satisfies weak completeness, repentance, and makes no further mistakes.

By (b) and (d), $\mathcal{SF}(m)$ satisfies the specification of a Strongly m-Mistaken failure detector in run R_A. From (c), R_A, a run of A that uses $\mathcal{SF}(m)$, violates the specification of Consensus. □

We now show that the above lower bound is tight: Given $\mathcal{SF}(m)$, Consensus can be solved in asynchronous systems with $m \leq n - f$.

Theorem 41: If $m \leq n - f$ then Consensus can be solved in asynchronous systems using any Strongly m-Mistaken failure detector $\mathcal{SF}(m)$.

PROOF: Suppose $m < n - f$. Since m, the number of mistakes made by $\mathcal{SF}(m)$, is less than the number of correct processes, there is at least one correct process that $\mathcal{SF}(m)$ never suspects. Thus, $\mathcal{SF}(m)$ satisfies weak accuracy. By definition, $\mathcal{SF}(m)$ also satisfies weak completeness. So $\mathcal{SF}(m)$ is a Weak failure detector and can be used to solve Consensus (Corollary 15).

Suppose $m = n - f$. Even though $\mathcal{SF}(m)$ can now make a mistake on *every* correct process, it can still be used to solve Consensus (even if a majority of the processes are faulty). The algorithm uses rotating coordinators, and is similar to the one for $\diamond\mathcal{W}$ in

Figure 6. Because of this similarity, we omit the details from this Appendix. □

From the above two theorems:

Corollary 42: Suppose $f \geq \frac{n}{2}$. Consensus can be solved in asynchronous systems using any $\mathcal{SF}(m)$ if and only if $m \leq n - f$.

We now turn our attention to Weakly k-Mistaken failure detectors.

Theorem 43: Suppose $f \geq \frac{n}{2}$. If $m > 0$ then there is a Weakly m-Mistaken failure detector $\mathcal{WF}(m)$ such that there is no algorithm A which solves Consensus using $\mathcal{WF}(m)$ in asynchronous systems.

PROOF: In Theorem 40, we described a failure detector that cannot be used to solve Consensus in asynchronous systems with $f \geq \frac{n}{2}$. It is easy to verify that this failure detector makes at most one mistake about each correct process, and thus it is a Weakly m-Mistaken failure detector. □

From Corollary 37 and the above theorem, we have:

Corollary 44: Suppose $f \geq \frac{n}{2}$. Consensus can be solved in asynchronous systems using any $\mathcal{WF}(m)$ if and only if $m = 0$.